A Trace-Based Systems Framework : Models, Languages and Semantics

Lotfi S. SETTOUTI, Yannick PRIÉ
Pierre-Antoine CHAMPIN, Jean-Charles MARTY, Alain MILLE

February 27, 2009

1 Introduction

If computer is going to become a new medium to think with and work, if we are going to be able to interact and communicate in multiple modalities, an efficient and intuitive means of supporting our use and interaction with computer must be developed. We specially need ways of encoding mechanisms to support user interactions, involved in an activity inherently temporal and dynamic, as web navigation and search, eLearning activity, etc. Such mechanisms will allow human and computational agents to parse, process and assist users in their complex and dynamic activities.

However, if we want to design such a system capable of behaving *intelligently* in some environment, then we need obviously to supply this system with mechanisms to declare, evolve and improve the sufficient knowledge to observe, interpret and reason about its observations of this environment. To do that, we need an unambiguous language capable of expressing this knowledge, together with some precise and well understood way of manipulating sets of observations which allow us to draw inferences, answer queries, make interpretations and update both the knowledge base and the desired system behaviour.

In this paper, the main contribution is to consider traces of user computer interaction as system's knowledge of user activities and experiences. As a knowledge-based system (KBS) exploits an explicit representation of different kinds of knowledge, we describe Trace-Based System (TBS) as a kind of KBS whose main source of knowledge is the set of trace subsuming user-system interactions and evolving with his/her activities. The remainder of the paper is organized as follows. Section 2 presents the general architecture of our framework describing the several services offered by TBS. Sections (5-6-7) present a formal representation of the concept of *modelled traces* so called and the associated languages supporting reasoning about them and their interpretation. Firstly, we focus on formalization of trace model and $\mathcal{M}$-Trace in offline exploitation. Then, we specify with precise semantics and based on $\mathcal{M}$-Traces, a languages to describe patterns, queries, transformations. Section 3 extends this

formalisation to the case of online exploitation by defining the notion of online $\mathcal{M}$-Traces and continuous transformations and queries. Section 9 describes the implementation of our framework by defining a translation of our concept into the logic-based language datalog. Section 10 we discuss our proposition and the future works.

2 Trace-Based Systems Framework

Although it seems to have become common sense that *traces* are important matters for observing user interaction, there is no shared understanding of what a *trace* is. In this paper, we consider interaction traces as *a sequence of observed elements* recorded from a user's interaction and navigation through a specific system. The term *Sequence* refers to an existing order relation representing a history of the user's interaction process during the observed activity. *Observed Elements* indicate that the trace data result from an observation. Such elements could represent an action, a message, a object, etc.

We assume that trace-based systems use an abstract architecture as shown in Figure 1. At the top of the general architecture of a TBS is the tracing system, which captures or collects the observed data from different input sources (log files, streamed actions, video records, interface events, etc.). The tracing system elaborates so called primary traces (often low level) from active or passive tracing sources. Conceptually, observed elements are stored in two partitions: working storage containing on-line traces captured from active tracing sources and persistent storage for off-line traces collected from several passive tracing sources or stored after the observation.

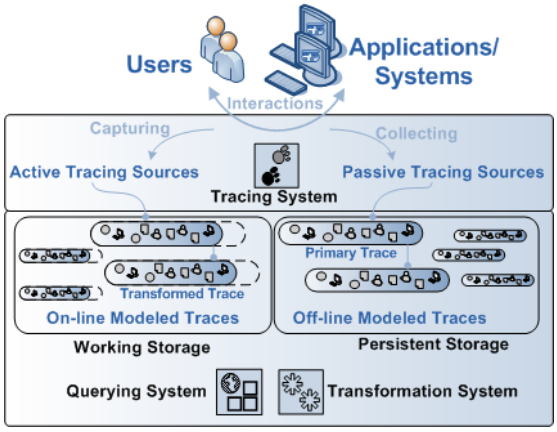

Figure 1: Trace-Based System Architecture

A Transformation System can perform operations on traces like applying filters, rewriting and aggregating elements, computing elements attributes, etc. so as to produce so called transformed modelled traces that can be more easily

reusable and exploitable in a given context than primary traces. Transformations can also be considered as semantic abstractions when associated with models (ontologies) like task model. The Querying System enables the extraction of episodes and patterns from the traces.

Users or systems may query, transform off-line or on-line traces. The working storage is persisted automatically on demand or periodically. Persistent queries and transformations are registered in the transformation or querying systems. Results of queries or transformations may be streamed to users or another system or materialized in the TBS.

To define and implement such a system, we have defined languages to model, query and transform $\mathcal{M}$-Traces. Modelling, representing and processing trace elements involve having a precise language with well-defined semantics. As explicit semantic admits a wide variety of forms, we have formally defined the concepts of trace and trace-model avoiding ambiguities in reasoning, computing, resolving queries and transformations in off-line and on-line exploitations.

Thus, our approach towards common model and semantics uses a declarative language with well-defined semantics for expressing modelled traces, queries and transformations. Our formal framework is described in details in the next section.

3 A Formal Framework of Trace-Based Systems

Let S be an ordered set, $\mathbf{I}(S)$ is the set of finite intervals on S. Let I be an interval, $\inf(I)$ is the greatest lower bound of I, $\sup(I)$ is the lowest upper bound of I.We assume that there exist a set $\mathbf{V}$ of literal values (sometimes called concrete values), and a set $\mathbf{D}$ of datatypes. Each datatype $d \in \mathbf{D}$ has a value-space $V_d \subseteq \mathbf{V}$. An example of such sets are the ones defined by XML-Schemas [25]. For convenience, we will not distinguish in this document between the datatype and its value space, hence $v \in d$ will mean that $v \in \mathbf{V}$ belongs to the value-space of $d \in \mathbf{D}$.

4 Representing time

Traces are about time-situated observations, hence they require a representation of time. In this section we define the formal notions we need to represent time in trace-based systems, and discuss how they are operationally used.

4.1 Formal representation

Different traces or trace models may have different representations of time. However, they all share a common ground, namely, that time can be seen as an order collection of discrete instants, such that every instant has a unique successor.

Definition 1 (Temporal Domain)**.** *A Temporal Domain $\mathcal{T}$ is a countable set of instants. A bijective function* $\text{succ}_{\mathcal{T}} : \mathcal{T} \to \mathcal{T}$ *associates to every instant its successor, and defines a total order* $\leq_{\mathcal{T}}$ *on* $\mathcal{T}$*.*

Inside a given temporal domain, a trace only describes a finite interval of time in that domain. We call such an interval the temporal extension of the trace.

Definition 2 (Temporal Extension)**.** *Given a temporal domain $\mathcal{T}$, a Temporal Extension $\mathcal{E}_{\mathcal{T}}$ is any element from* $\mathbf{I}(\mathcal{T})$*.*

4.2 Operational representation

In an operational Trace-Based System, we need to be able to identify different temporal domains and temporal extensions, to address the different requirements of different applications. More precisely, we differentiate between applications only requiring an order between instants, and applications requiring the notion of *duration* between instants. We call the first kind of temporal extensions *Sequential Temporal Extensions*, and the second kind *Chronometrical Temporal Extensions*.

4.2.1 Sequential Temporal Extensions

This kind of temporal extension implies nothing more than what is defined in definitions 1 and 2. However, for the sake of simplicity and without loss of generality, we will assume that all sequential temporal extensions belong a unique temporal domain named $\mathcal{T}_{\text{seq}}$.

4.2.2 Chronometrical Temporal Domains

In order to account for the notion of duration in chronometrical temporal extensions in a domain $\mathcal{T}$, we will assume that the duration between any instant $t \in \mathcal{T}$ and its successor $\text{succ}_{\mathcal{T}}(t)$ is a constant amount of time, or *unit*. Hence, the duration between two instants t and t' can be computed by simply counting the number of instants between t and t', and that measure is consistent across all temporal extensions from $\mathbf{I}(\mathcal{T})$. More precisely, for every temporal unit u, we define the temporal extension $\mathcal{T}_u$.

Note that we impose no restriction on the notion of unit, hence allow for different notions of time to cohabit under the notion of chronometrical temporal domain. For example, $\mathcal{T}_{day}$ and $\mathcal{T}_{month}$ are not directly commensurable, because all months do not have the same number of days, although it is possible to measure durations in days or in months. It is also important to note that comparing $\mathcal{T}_{day}$ and $\mathcal{T}_{hour}$ is not easy either, because events located in the former are nor measured precisely enough to be automatically located in the latter.

4.2.3 Chronometrical Temporal Extensions

Given a chronometrical temporal domain $\mathcal{T}_u$, we can specify any temporal extension by a triple (o, b, e) where $o \in \mathcal{T}_u$, $b \in \mathbf{N}$ and $e \in \mathbf{N}$. It represents the interval between the b^{th} and the e^{th} successors of o, or more formally the interval $[\text{succ}^b_{\mathcal{T}_u}(o), \text{succ}^e_{\mathcal{T}_u}(o)]$. Every instant in that interval can be represented by an integer $i \in [b, e]$, representing the i^{th} successor of o; i is called the *index* of the corresponding instant.

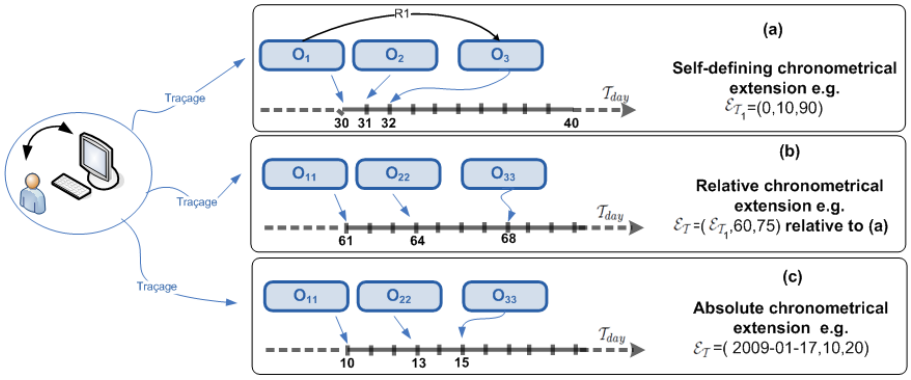

Figure 2: Chronometrical Temporal Extensions Example

We did not define yet how one may represent the origin o of a chronometrical temporal extension. They are actually three ways to do this:

- **Self-defining chronometrical extension.** An arbitrary name is given to the origin, not allowing to compare it to any instant from another extension (figure 2). Durations in that extension are still commensurable with durations in other extensions from the same domain, but its instants cannot be located relatively to those from that extension.

- **Relative chronometrical extension.** The arbitrary name from another extension is given for the origin, indicating that this extension is reusing a previously defined origin (figure 2). In all extensions using the same origin, not only durations but also instant indices can be meaningfully compared.

- **Absolute chronometrical extension.** The origin is specified according to a given standard, like for example [29]. This allows us to compare two origins. Hence, in all absolute extensions from the same domain, durations and indices can be compared even if they do not have the same origin (figure 2).

5 $\mathcal{M}$-Traces and $\mathcal{M}$-Trace model

Before we define the notion of $\mathcal{M}$-Trace (short for *modelled* trace), we need to define the notion of a trace model.

Definition 3 (Trace Model)**.** *A Trace Model is defined as a tuple*

$$\mathcal{M}_{\mathbb{T}^r} = (\mathcal{T}, C, R, A, \mathrm{dom}_R, \mathrm{range}_R, \mathrm{dom}_A, \mathrm{range}_A)$$

consisting of

- *a temporal domain* $\mathcal{T}$*,*
- *a finite set* C *of observed element types (or classes), with a partial order* $\leq_C$ *defined on it,*
- *a finite set* R *of relation types, disjoint from* C*, with a partial order* $\leq_R$ *defined on it,*
- *a finite set* A *of attributes, disjoint from* C *and* R*,*
- *two functions* $\mathrm{dom}_R : R \to C$ *and* $\mathrm{range}_R : R \to C$ *defining the domain range of relation types,*
- *two functions* $\mathrm{dom}_A : A \to C$ *and* $\mathrm{range}_A : A \to \mathbf{D}$ *defining the domain and range of attributes,*

It must also hold for any two relations r_1 *and* r_2 *that*

$$r_1 \leq_R r_2 \Rightarrow \mathrm{dom}_R(r_1) \leq_C \mathrm{dom}_R(r_2) \wedge \mathrm{range}_R(r_1) \leq_C \mathrm{range}_R(r_2)$$

Intuitively, a trace model defines a vocabulary for describing traces: how time is represented ($\mathcal{T}$), how observed elements are categorized (C), what relations may exist between observed elements (R), what attributes further describe each observed elements (A). The domain and range function constrain the kind of relations and attributes that an observed element of a given type may have. Partial orders $\leq_C$ and $\leq_R$ induce a type hierarchy for observed elements and relations. The last constraint guarantees the consistency of domain and range between a relation and its parents in the hierarchy. To exemplify our model in the context of web navigation observation., we present in the sequel a trace model instanciated by a key-logger that we have developed.

It is difficult to force a user to perform the actions required to create history trace such as when and how he or she viewed a web page. So we developed a tracking module that automatically saves detailed traces of the user's interaction. The automatic module monitors the event messages of an operation system (OS) and it does not depend on specific applications. Specifically, it collects the computer's mouse, keyboard, copying, and printing event and window conditions, etc. Figure 3 shows the entire trace model used by the tracking module[1].

[1]The tracking has also an encryption function to protect the user's privacy.

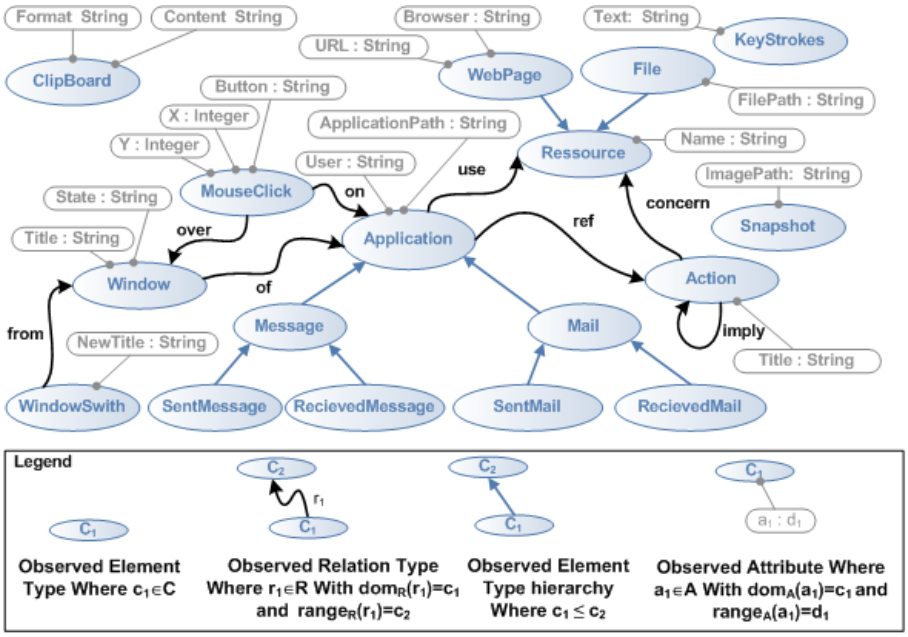

Figure 3: An example of Trace Model

In the rest of this papier, for the sake to describe and demonstrate all possibilities supported by our framework, we will use in some examples only a subsect of this $\mathcal{M}$-Trace model or models resulting from transformations starting from traces conforming to this model.

Definition 4 ($\mathcal{M}$-Trace)**.** *A $\mathcal{M}$-Trace is a tuple*

$$\mathbb{T}^r = (\mathcal{M}_{\mathbb{T}^r}, \mathcal{E}_{\mathcal{T}}, O, id, \lambda_C, \lambda_R, \lambda_A, \lambda_T)$$

consisting of

- *a trace model* $\mathcal{M}_{\mathbb{T}^r} = (\mathcal{T}, C, R, A, \mathrm{dom}_R, \mathrm{range}_R, \mathrm{dom}_A, \mathrm{range}_A)$,
- *a temporal extension* $\mathcal{E}_{\mathcal{T}}$,
- *a finite set* O *of observed elements, disjoint from* C, R *and* A
- *an invertible total*[2] *function* $id : O \to \mathbf{V}$ *(i.e.* id *is bijective between* O *and* $\mathrm{range}(id)$*),*
- *a total function* $\lambda_C : O \to C$ *called element type labeling,*
- *a relation* $\lambda_R \subseteq O \times O \times R$ *called relation type labeling,*

[2] $id(o)$ is defined on *every* $o \in O$

- *a partial*[3] *function* $\lambda_A : O \times A \rightarrow \mathbf{V}$ *called attribute labeling,*
- *a total function* $\lambda_T : O \rightarrow \mathbf{I}(\mathcal{E}_\mathcal{T})$ *called time labeling.*

Furthermore, a trace must be consistent *to its model,* i.e. *verify the following constraints:*

- $\mathcal{E}_\mathcal{T} \in \mathbf{I}(\mathcal{T})$
- $\forall (o_1, o_2, r) \in \lambda_R,\ \lambda_C(o1) \leq_C \text{dom}_R(r) \wedge \lambda_C(o2) \leq_C \text{range}_R(r)$
- $\forall (o, a, v) \in \lambda_A,\ \lambda_C(o) \leq_C \text{dom}_A(a) \wedge v \in \text{range}_A(a)$

As a convenient notation, we will sometimes use λ_R as a function such that $\lambda_R(o_1, o_2) = \{r, (o_1, o_2, r) \in \lambda_R\}$.

Intuitively, an $\mathcal{M}$-Trace represents, according to a trace model ($\mathcal{M}_{\mathbb{T}^r}$), a given period of observation ($\mathcal{E}_\mathcal{T}$). It contains a set of typed observed elements (O and λ_C), each with a unique identifier (id), located in time (λ_T), in relation with each other (λ_R), and described by attribute values (λ_A). Each observed element o has exactly one *direct* type (λ_C is a total function); note that the relation $\leq_C$ induces a kind of type inheritance, so every type $c \geq_C \lambda_C(o)$ may be considered an *indirect* type of o. There may be no, one or several relation(s) between two given observed elements (λ_R can be any relation). Finally, attribute values are never mandatory. The $\mathcal{M}$-Trace is consistent with its model if its temporal extension actually belongs to the model's temporal domain, and if domain and range constraints on relations and attributes are all satisfied. The figure 4 will examplify a portion of an $\mathcal{M}$-Trace having a sub-model of the $\mathcal{M}$-Trace model presented in precendent definition.

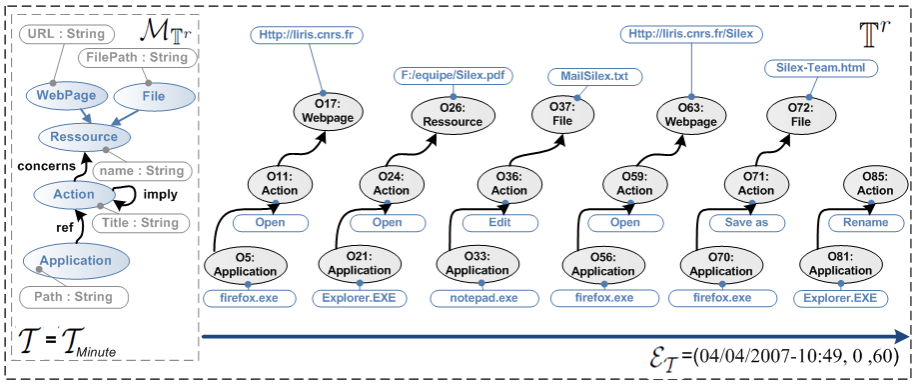

Figure 4: An example of simplified keylogger $\mathcal{M}$-Trace

[3] $\lambda_A(o, a)$ may be undefined for some $(o, a) \in O \times A$

6 $\mathcal{M}$-Trace Pattern and Query Language

In this section, we introduce the notions necessary to querying an $\mathcal{M}$-Trace, *i.e.* identifying in that $\mathcal{M}$-Trace a set of observed elements satisfying a number of criterions. Those critierions are expressed in a pattern.

In the following, we will assume that there is an infinite set ϑ of variable, disjoint will all the sets introduced in $\mathcal{M}$-Tracesand trace models.

6.1 Pattern Syntax and Query

Definition 5 ($\mathcal{M}$-Trace Alphabet and Pattern)**.** *For a given trace $\mathbb{T}^r$, we define the pattern alphabet $\Sigma_{\mathbb{T}^r}$ as follow:*

$$\Sigma_{\mathbb{T}^r} = \vartheta \cup \mathcal{T} \cup C \cup R \cup A \cup \mathbf{V} \cup O \cup N \cup K$$

where $\vartheta, \mathcal{T}, C, R, A, \mathbf{V}, O, N$ and K are pairwise disjoint; $\mathcal{T}, C, R, A$ and O are the temporal domain, set of types, relations, attributes and observed elements defined by the trace and its model; N is an infinite set of query names; and K is the set of keywords, i.e. *all the terminal symbols between double quotes in the grammar below.*

We define an $\mathcal{M}$-Trace pattern *on $\mathbb{T}^r$ as any word on $\Sigma_{\mathbb{T}^r}$ matching $< P >$ in the following grammar:*

$$\begin{array}{rcl}
< P > & ::= & o \text{ ":" } c \mid o\ r\ o \mid \mathcal{AE}(< AT >)\ < Cp >\ \mathcal{AE}(< AT >) \\
 & & \mid\ < P >\ \text{","}\ < P > \mid \text{"("}\ < P >\ \text{")"} \\
 & & \mid\ < PX > \\
< AT > & ::= & v \mid o\text{"."}a \mid t \mid o\text{".begin"} \mid o\text{".end"} \mid o\text{".id"} \\
< Cp > & ::= & \text{"}=\text{"} \mid \text{"}\neq\text{"} \mid \text{"}<\text{"} \mid \text{"}\leq\text{"} \mid \text{"}\geq\text{"} \mid \text{"}>\text{"} \\
< PX > & ::= & < W > \mid < GP > \\
< PX > & ::= & < Q > \mid < W > \mid < GP > \\
< Q > & ::= & q\ \text{"("}\ < params >\ \text{")"} \\
< params > & ::= & o\ \text{","}\ < params > \mid o \\
< W > & ::= & < P >\ \text{"}without\text{"}\ \text{"\{"}\ < P >\ \text{"\}"} \\
< GP > & ::= & < P >\ \text{"}or\text{"}\ < P > \mid < P >\ \text{"}opt\text{"}\ < P > \\
 & & \\
 with & & o \in O \cup \vartheta,\ c \in C, r \in R, v \in V, a \in A, t \in \mathcal{T}, x \in \vartheta, q \in N
\end{array}$$

$\mathcal{AE}(< AT >)$ is a generic production rule for arithmetic expressions evaluating to a value where terms match the production rule $< AT >$.

Note that we didn't commit to explicitly define the syntax of arithmetic expressions. It will depend on the actual set of datatypes used by applications. We envision that implementations may authorize basic operators, but also standard functions on numbers (*cos*, *sin*...), strings (*substring*, regular expression matching...) or instants (substraction for chrometrical temporal domains, conversion from one domain to another...). It is not the purpose of this document to exhaustively describe such epressions, but rather to focus on aspects that are specific to $\mathcal{M}$-Traces.

Example 1. *Let we consider the $\mathcal{M}$-Trace $\mathbb{T}_1^r$ examplified in section 5. We can express the following pattern*
P_1=`((X: Application),(Y: Action),(X ref Y),(Y.Title="Save As"))`
The result of evaluation of pattern P_1 is obviously $\{X = O_{70}, Y = O_{71}\}$.

The next section will present the semantics of such pattern evaluation. Foremost, we need to describe some notions required to define the pattern semantics. Let P be an $\mathcal{M}$-Trace pattern. The set of *bound variables*, noted $\mathrm{b}^{\vartheta}(P)$, is the set of all variables appearing only inside curly brackets (in production rule $< W >$). The set of *free variables*, noted $\mathrm{f}^{\vartheta}(P)$, is the set of all variables appearing in P that are not bound. If $\mathrm{f}^{\vartheta}(P) = \emptyset$, P is said to be *grounded*, else P is said to be *ungrounded*.

The fact that a pattern is linked to a specific $\mathcal{M}$-Trace and possibly contains references to observed elements of that trace may seem strange since we introduced patterns in order to represent queries, which should be usable on several $\mathcal{M}$-Traces. Indeed, the definition of a pattern above is more general that what we need for describing queries, but we will require it in the following to define the semantics of patterns and queries. The specific kind of patterns that will be used in queries is defined below.

Definition 6 (Model Pattern)**.** *An $\mathcal{M}$-Trace pattern is a* model pattern *if and only if it contains no symbol from O, the set ot observed elements, and has at least one free variable*[4]*.*

It is important to note that a model pattern does not depend anymore on a particular trace $\mathbb{T}^r$, but only on its model $\mathcal{M}_{\mathbb{T}^r}$, since it is defined over the vocabulary

$$\Sigma_{\mathcal{M}_{\mathbb{T}^r}} = (\vartheta \cup C \cup R \cup A \cup V \cup N \cup K)$$

We are now ready to define a query.

Definition 7 ($\mathcal{M}$-Trace Query)**.** *A* Query *on an $\mathcal{M}$-Trace-model $\mathcal{M}_{\mathbb{T}^r}$ is defined as a tuple*

$$\mathcal{Q} = (n, \vartheta_{\mathcal{Q}}, P_{\mathcal{M}})$$

where

- *n is the name of the query,*
- *$P_{\mathcal{M}}$ is a model pattern on $\mathcal{M}_{\mathbb{T}^r}$,*
- *$\vartheta_{\mathcal{Q}} \subseteq \mathrm{f}^{\vartheta}(P_{\mathcal{M}})$ is an ordered set of distinguished variables of $\mathcal{Q}$,*

It is assumed that every query will have a distinct name n, so that this name can be later used in pattern production rule $< Q >$ to identify the query. It is possible for the pattern of a query to reference the name of that same query.

Example 2. *We can define an $\mathcal{M}$-Trace $\mathcal{Q}_1$ using the pattern P_1.*
P_1=`((X: Application),(Y: Action),(X ref Y),(Y.Title="Save As"))`
$\mathcal{Q}_1$ =`(SavingActions, Y,` P_1`)`
The result of evaluation of query $\mathcal{Q}_1$ is $\{Y = O_{71}\}$.

[4] A trivial pattern, such as $(1 = 1)$ or $(1 \neq 1)$, is not a model pattern.

6.2 Semantics

In this section, we will need to distinguish two kinds of patterns, based on the production rules they use: basic patterns, and general patterns. Intuitively, a query is basic if its pattern does not use the *or* and *opt* operators.

Definition 8 (Basic and General Pattern)**.** *A pattern P is* basic *if and only if it does not use production rule $< GP >$. A pattern which is not basic is called a* general *pattern.*

We will first describe how the results of a query are represented. Then, we will define the semantics of querying a basic pattern. Finally we will define the semantics of querying a general pattern. Note that, since basic patterns may embed general patterns, the two semantics will be recursively interdependent.

6.2.1 Substitution and result sets

To formally define pattern and query semantics, we need the notion of *substitution*. We call a substitution function, or simply substitution, any function (partial or total) $\psi : \vartheta \to O$, where O is the set of observed element of an $\mathcal{M}$-Trace. Substitutions will be useful to define the semantics of patterns. For convenience, we will sometimes apply a substitution function directly to a pattern P: $\psi(P)$ will then denote a copy of P where each symbol $v \in \mathrm{f}^{\vartheta}(P)$ has been replaced by $\psi(v)$ if defined.

Substitutions will also be useful to represent results of a query: indeed, a substitution defines a *mapping* or *affectation* of a set of variables to a set of observed elements. The domain of that function (*i.e.* the set of variables on which it is defined) is noted $\mathrm{dom}(\psi)$.

The result of querying an $\mathcal{M}$-Trace $\mathbb{T}^r$ with a pattern P is therefore a *set of substitutions* of the form $\psi : \mathrm{f}^{\vartheta}(P) \to O$, and will be noted $[\![P]\!]_{\mathbb{T}^r}$. The formal semantics of this notation will be described in the two following section, first for basic patterns (definition 10), then for general patterns (definition 13).

The result of applying a query $\mathcal{Q} = (n, \vartheta_{\mathcal{Q}}, P_{\mathcal{M}})$ to an $\mathcal{M}$-Trace $\mathbb{T}^r$ is obtained by restricting[5] every result for $P_{\mathcal{M}}$ to the set of distinguished variables of $\mathcal{Q}$. We will therefore use the following shortcut notation

$$[\![\mathcal{Q}]\!]_{\mathbb{T}^r} \doteq \{\psi_{|\vartheta_{\mathcal{Q}}} \mid \psi \in [\![P_{\mathcal{M}}]\!]_{\mathbb{T}^r}\}$$

6.2.2 Querying a basic pattern

We are now ready to define the semantics of a basic pattern, through the notion of entailment.

Definition 9 (Basic Pattern Entailment)**.** *Considering an $\mathcal{M}$-Trace pattern P for a trace $\mathbb{T}^r = (\mathcal{M}_{\mathbb{T}^r}, \mathcal{E}_{\mathcal{T}}, O, \lambda_C, \lambda_R, \lambda_A, \lambda_T)$ and its model $\mathcal{M}_{\mathbb{T}^r} = (\mathcal{T}, C, R, A,$ $\mathrm{dom}_R, \mathrm{range}_R, \mathrm{dom}_A, \mathrm{range}_A)$, we will say that the trace $\mathbb{T}^r$* entails *P, noted $\models_{\mathbb{T}^r} P$ if and only if either:*

[5] The restriction of a function $f : A \to B$ to a subset X of A is the function $f_{|X} : X \to B$ such that $f_{|X}(x) = f(x)$ for all $x \in A$.

- *P is ungrounded, and there exist a substitution ψ such that $\models_{\mathbb{T}^r} \psi(P)$*
- *P is grounded, and either*
  - *P has the form $o\ :\ c$ and $\lambda_C(o) \leq_C c$*
  - *P has the form $o_1\ r\ o_2$ and $\exists r' \in \lambda_R(o_1, o_2) \wedge r' \leq_R r$*
  - *P has the form $\mathcal{AE}(< AT >)$ and the expression evaluates to True, with the following semantics for terms:*
    - *v itself for terms of the form (v) (literal values)*
    - *$\lambda_A(o, a)$ for terms of the form $(o.a)$*
    - *$\inf(\lambda_T(o))$ for terms of the form $(o.begin)$*
    - *$\sup(\lambda_T(o))$ for terms of the form $(o.end)$*
    - *$id(o)$ for terms of the form $(o.id)$*

    Should one of the terms be undefined (e.g. $\lambda_A(o, a)$), the expression is assumed to evaluate to False.
  - *P has the form $q(o_1, ..., o_n)$ and there is a query $\mathcal{Q} = (q, \vartheta_{\mathcal{Q}}, P')$ such that $\vartheta_{\mathcal{Q}} = \{v_1, ..., v_n\}$ and $\exists \psi \in [\![\mathcal{Q}]\!]_{\mathbb{T}^r},\ \forall i \leq n,\ \psi(v_i) = o_i$*
  - *P has the form $P',\ P''$ and $\models_{\mathbb{T}^r} P' \wedge \models_{\mathbb{T}^r} P''$*
  - *P has the form P' without $\{P''\}$ and $\models_{\mathbb{T}^r} P'\ \wedge\ \not\models_{\mathbb{T}^r} P''\}$*
  - *P has the form (P') and $\models_{\mathbb{T}^r} P'$*

We are now ready to define the result set of querying a basic pattern.

Definition 10 (Result Set of Querying a Basic Pattern)**.** *Let P be a basic pattern on $\mathcal{M}$-Trace model $\mathcal{M}_{\mathbb{T}^r}$, $\mathbb{T}^r$ an $\mathcal{M}$-Trace consistent with $\mathcal{M}_{\mathbb{T}^r}$ and O its set of observed elements. The result set of querying $\mathbb{T}^r$ with P is defined as follow:*

$$[\![P]\!]_{\mathbb{T}^r} = \{\psi : \mathrm{f}^{\vartheta}(P) \to O \mid \ \models_{\mathbb{T}^r} \psi(P)\}$$

Note that the results for a basic pattern will always match *all* of its free variables. This will not be the case with general pattern.

6.2.3 Querying a general pattern

Unlike basic patterns, the semantics of querying a general pattern can not easily be defined with the notion of entailment. To properly define it, we need to extend the semantics of basic pattern querying by formalizing operations among result sets (which contain substitution functions).

Definition 11 (Compatible Substitutions)**.** *Two substitutions ψ_1 and ψ_2 are compatible if and only if they substitute the same values to their common variables,* i.e. $\forall x \in \mathrm{dom}(\psi_1) \cap \mathrm{dom}(\psi_2), \psi_1(x) = \psi_2(x)$. *We note* $\mathrm{comp}(\psi_1, \psi_2)$.

Note that two substitutions with disjoint domains are always compatible, and that the empty substitution (*i.e.* the substitution with empty domain) $\psi_\varnothing = \varnothing$ is compatible with any other substitution. Note also that if ψ_1 and ψ_2 are compatible, then $\psi_1 \cup \psi_2$ is also a substitution.

Definition 12 (Result Set Operators)**.** *Let φ_1 and φ_2 be two result sets. We define the three operator inner join (noted $\bowtie$), compatible difference (noted $\ominus$) and left outer join (noted $\rtimes$) as:*

- $\varphi_1 \bowtie \varphi_2 = \{\psi_1 \cup \psi_2 \mid \psi_1 \in \varphi_1 \wedge \psi_2 \in \varphi_2 \wedge \text{comp}(\psi_1, \psi_2)\}$,
- $\varphi_1 \ominus \varphi_2 = \{\psi_1 \in \varphi_1 \mid \forall \psi_2 \in \varphi_2,\ \neg\text{comp}(\psi_1, \psi_2)\}$.
- $\varphi_1 \rtimes \varphi_2 = (\varphi_1 \bowtie \varphi_2) \cup (\varphi_1 \ominus \varphi_2)$

Finally, we can now define the semantics of general pattern querying.

Definition 13 (Result Set of Querying a General Pattern)**.** *Let P be a general pattern on $\mathcal{M}$-Trace model $\mathcal{M}_{\mathbb{T}^r}$, $\mathbb{T}^r$ an $\mathcal{M}$-Traceconsistent with $\mathcal{M}_{\mathbb{T}^r}$ and O its set of observed elements. The result set of querying $\mathbb{T}^r$ with P is defined as follow:*

- *if P has the form of basic pattern, $[\![P]\!]_{\mathbb{T}^r}$ as defined in definition 10;*
- *if P has the form P', P'', $[\![P]\!]_{\mathbb{T}^r} = [\![P']\!]_{\mathbb{T}^r} \bowtie [\![P'']\!]_{\mathbb{T}^r}$;*
- *if P has the form P' without $\{P''\}$, $[\![P]\!]_{\mathbb{T}^r} = [\![P']\!]_{\mathbb{T}^r} \ominus [\![P'']\!]_{\mathbb{T}^r}$;*
- *if P has the form P' or P'', $[\![P]\!]_{\mathbb{T}^r} = [\![P']\!]_{\mathbb{T}^r} \cup [\![P'']\!]_{\mathbb{T}^r}$;*
- *if P has the form P' opt P'', $[\![P]\!]_{\mathbb{T}^r} = [\![P']\!]_{\mathbb{T}^r} \rtimes [\![P'']\!]_{\mathbb{T}^r}$;*
- *if P has the form (P'), $[\![P]\!]_{\mathbb{T}^r} = [\![P']\!]_{\mathbb{T}^r}$*

Example 3. *Take for instance the $\mathcal{M}$-Trace examplified in figure 4 noted $\mathbb{T}_1^r$ and following general patterns*
P_2=`((X: Action),(X concerns Y),(Y: File))`
P_3=`((X.Title='Open'),(Z ref X),(Z.Path='firefox.exe'))`
P_4=`((X: Action) without { (Y.Title='Open'),(Y.end < X.begin) })`
P_5=`(((X.Title='Open'),(Z ref X)) OPT (Z.Path='Explorer.exe'))`
P_6=`(`P_2 `OR` P_4`)`

Then, when viewing each solution set as a table with variables denoting attribute names, we can write:

$[\![P_2]\!]_{\mathbb{T}_1^r}=$

X
O_{11}
O_{24}
O_{36}
O_{59}
O_{71}
O_{85}

$\bowtie$

X	Y
O_{11}	O_{17}
O_{24}	O_{26}
O_{36}	O_{37}
O_{59}	O_{63}
O_{71}	O_{72}

$\bowtie$

Y
O_{37}
O_{72}

$=$

X	Y
O_{36}	O_{37}
O_{71}	O_{72}

$[\![P_3]\!]_{\mathbb{T}_1^r}=$

X
O_{11}
O_{24}
O_{59}

$\bowtie$

Z	X
O_5	O_{11}
O_{21}	O_{24}
O_{33}	O_{36}
O_{56}	O_{59}
O_{70}	O_{71}
O_{81}	O_{85}

$\bowtie$

Z
O_5
O_{56}

$=$

X	Z
O_{11}	O_5
O_{59}	O_{56}
O_{71}	O_{70}

$[\![P_4]\!]_{\mathbb{T}_1^r}=$

X
O_{11}
O_{24}
O_{36}
O_{59}
O_{71}
O_{85}

$\ominus$

Y	X
O_{11}	O_{24}
O_{11}	O_{59}
O_{24}	O_{59}

$=$

X
O_{11}
O_{36}
O_{71}
O_{85}

$[\![P_5]\!]_{\mathbb{T}_1^r}=$

X	Z
O_{11}	O_5
O_{24}	O_{21}
O_{59}	O_{56}

$\rtimes$

Z
O_{21}
O_{81}

$=$

X	Z
O_{24}	O_{21}
O_{11}	O_5
O_{59}	O_{56}

($\rightarrow$ Resulting from $\rtimes$)
($\rightarrow$ Resulting from $\ominus$)
($\rightarrow$ Resulting from $\ominus$)

$[\![P_6]\!]_{\mathbb{T}_1^r}=$

X	Y
O_{11}	O_{17}
O_{59}	O_{63}

$\cup$

X
O_{11}
O_{36}
O_{71}
O_{85}

$=$

X	Y
O_{11}	O_{17}
O_{59}	O_{63}
O_{36}	-
O_{71}	-
O_{85}	-

It is interesting to note that the substitutions in the result set of querying a general pattern, unlike those of a basic pattern, may not be total on the set of free variables of P. Consider for example pattern $(x : C_1 \ or \ y : C_2)$, where the set of free variables is $\{x, y\}$. Its result set will contain substitutions defined either on $\{x\}$ or $\{y\}$, since it is constructed as the union of the result sets of the sub-patterns. Now consider the pattern $(x : C \ opt \ x \ R \ y)$, where the set of free variables is $\{x, y\}$. Its results will be defined either on $\{x\}$ (for those where no match where found for the optional sub-pattern) or $\{x, y\}$ (for those where there was a match).

Such partial results correspond to the case, in other query languages, where some variables are assigned a *null* values.

7 $\mathcal{M}$-Trace Transformations

In this section, we introduce the concept of *transformation*. A transformation defines a means to produce a new $\mathcal{M}$-Trace from one or several others, or *transform* the original set into a new one. Note that the original traces are not actually modified. Applying a transformation τ to a sequence of traces $(\mathbb{T}_i^r)$ associated to trace models $\mathcal{M}_{\mathbb{T}^r}^i$ will produce a new transformed trace $\mathbb{T}_\tau^r$ with a target trace model $\mathcal{M}_{\mathbb{T}^r}^\tau$, and will be noted:

$$(\mathbb{T}_1^r, ..., \mathbb{T}_n^r) \xrightarrow{\tau} \mathbb{T}_\tau^r$$

We will define three particular kinds of transformation, that are expected to cover the main uses in trace based systems.

- an *elementary transformation* applies to a single source $\mathcal{M}$-Trace. Such a transformation is constrained by two trace models (source and target) and any trace consistent with the source trace model $\mathcal{M}_{\mathbb{T}^r}$ may be transformed

to a novel trace that will be consistent with the target trace model $\mathcal{M}^{\tau}_{\mathbb{T}^r}$ (see definition 4 for description of $\mathcal{M}$-Trace consistence).

- a *composite transformation* is a graph of elementary or fusion/concatenation transformations, having an arbitrary number of sources and exactly one pit.
  - a *fusion transformation* applies to a set of $\mathcal{M}$-Traces all consistent with a unique trace model. It produces a transformed $\mathcal{M}$-Trace with the same model; its temporal extension is the smallest temporal extension containing all the temporal extensions of the source $\mathcal{M}$-Traces; it contains a copy of all observed elements and relations from the source traces.
  - a *concatenation transformation* applies to a sequence of $\mathcal{M}$-Traces all consistent with a unique trace model, and with a *sequential* temporal extension. It produces a transformed $\mathcal{M}$-Trace with the same model, and containing a copy of all observed elements and relations from the source traces.

In this paper, we focus our formalisation on elementary transformations. However, we plan to extend our formalisation to composite transformation in a future work taking into account the presented complexity evaluation of elementary transformation.

7.1 Elementary Transformation Rule Syntax

Informally, an elementary transformation must describe two parts to build the new transformed $\mathcal{M}$-Trace.

- a set of transformation rules to fill the new $\mathcal{M}$-Trace. Informally, the tranformation rules are specified by means of *patterns* and *template* where pattern describes a set of observed elements which will be transformed and used by template to construct the $\mathcal{M}$-Trace.
- optionnaly one final rule to calculate the temporal extension (i.e. a computation from the input-trace's temporal extension). If such rule is not defined then the trace temporal extension must be (min, max) of time of created observed elements

We first define formally a transformation rule.

Definition 14 (Transformation Rule)**.** *Let $\mathcal{M}_{\mathbb{T}^r}, \mathcal{M}'_{\mathbb{T}^r}$ be two $\mathcal{M}$-Trace models. We define an* $\mathcal{M}$-Trace transformation rule *from $\mathcal{M}_{\mathbb{T}^r}$ to $\mathcal{M}'_{\mathbb{T}^r}$ as a tuple of the form $(P_{\mathcal{M}}, G)$, where*

- *$P_{\mathcal{M}}$ is an model pattern on $\mathcal{M}_{\mathbb{T}^r}$,*
- *G is an model pattern on $\mathcal{M}'_{\mathbb{T}^r}$, named the* template *of the rule.*

The variables occuring in $P_{\mathcal{M}}$ are called the pattern variables*, while those occuring only in G are called* template variables*. Furthermore, G must satisfy the following constraints:*

- *production rule $< PX >$ is not used.*
- *every variable w not appearing in P must at least appear in sub-patterns of each following form:*
  - $w : c$ *with* $c \in C'$
  - $w.begin = \mathcal{AE}(< AT >)$
  - $w.end = \mathcal{AE}(< AT >)$
  - $w.id = \mathcal{AE}(< AT >)$

  where the arithmetic expressions must only contain pattern variables.

7.2 Elementary Transformations Semantics

Intuitively, a transformation rule associates a template (i.e. how we construct) to a specific model pattern (i.e. from what we construct). In fact, the pattern part is used to extract from source trace the observed elements which will be transformed, whereas the template part serves to produce a *fragments* of the target trace. Thus, to define the semantics of transformation rule, we need to define :

- the result set of pattern model (defined in 13),
- the result set of a template (i.e. how the template substitutes the result set of model pattern),
- the trace fragment produced by a transformation rule from template results.

The template can be evaluated on the result set of pattern model where all substitutions are defined (i.e. not null). The template can not be evaluated if it references undefined variables in the result sets of pattern. To define the semantic of template, we must specify the template substitution semantics. Unlike pattern substitution which replace every variable by observed elements according to defined $\mathcal{M}$-Trace, template substitution must remove, in addition, templates which references an undefined variables or attribute values.

Definition 15 (Template Substitution Semantics)**.** *Given a model pattern $P_{\mathcal{M}}$ associated to a template G defined over trace model $\mathcal{M}'_{\mathbb{T}^r}$. We define the result of applying a substitution $\psi \in [\![P_{\mathcal{M}}]\!]_{\mathbb{T}^r}$ according to $\mathbb{T}^r$, noted $\phi_\psi(G)$ as a set of ungrounded pattern obtained as follow:*

1. *replacing in each template every variable $x \in \mathrm{dom}(\psi)$ by $\psi(w)$, and removing every templates using irreplaceable variables not defined for ψ* i.e. *$\notin \mathrm{dom}(\psi)$;*

2. *replacing in each remaining templates every expression of the form of o.begin (respectively o.end) by* $\inf(\lambda_T(o))$ (*respectively* $\sup(\lambda_T(o))$*;*

3. *replacing in each template every expression of the form of o.a where* $o \in range(\psi)$ *by* $\lambda_A(o, a)$ *if defined, and removing every template using improper expression of the form o.a (*i.e. *in which* $\lambda_A(o, a)$ *is not defined);*

4. *removing all remaining templates implying a variable w for which the template of the* $(w : c)$ *has been remove.*

Then, the substitution of a template G over $P_{\mathcal{M}}$ *applied on* $\mathbb{T}^r$*, noted* $\phi_{P_{\mathcal{M}}}(G)$ *is defined as follow:*

$$\phi_{P_{\mathcal{M}}}(G) = \bigcup_{\psi \in [\![P_{\mathcal{M}}]\!]_{\mathbb{T}^r}} \phi_\psi(G)$$

Definition 16 (Transformation Rule Semantics)**.** *Let* $\mathbb{T}^r = (\mathcal{M}_{\mathbb{T}^r}, \mathcal{E}_{\mathcal{T}}, O, \lambda_C, \lambda_R, \lambda_A, \lambda_T)$*,* $\mathbb{T}^{r\prime} = (\mathcal{M}'_{\mathbb{T}^r}, \mathcal{E}'_{\mathcal{T}}, O', \lambda'_C, \lambda'_R, \lambda'_A, \lambda'_T)$ *be two* $\mathcal{M}$*-Traces. Let* $(P_{\mathcal{M}}, G)$ *be a transformation rule where* $P_{\mathcal{M}}$ *is a pattern model expressed on a trace model* $\mathcal{M}_{\mathbb{T}^r}$ *and G is a template using* $\mathcal{M}'_{\mathbb{T}^r}$*. Then, the result of applying transformation rule* $(P_{\mathcal{M}}, G)$ *from* $\mathcal{M}_{\mathbb{T}^r}$ *to* $\mathcal{M}'_{\mathbb{T}^r}$ *over an* $\mathcal{M}$*-Trace* $\mathbb{T}^r$*, noted* $[\![P_{\mathcal{M}}, G]\!]_{\mathbb{T}^r}$*, is defined as follow:*

$$[\![P_{\mathcal{M}}, G]\!]_{\mathbb{T}^r} = \bigcap \mathbb{T}^r_i \ where \ \models_{\mathbb{T}^{r\prime}} \phi_{P_{\mathcal{M}}}(G)\}$$

and

$$\bigcap \mathbb{T}^r_i \doteq \mathbb{T}^r, \forall x \in \ \pi_k(\mathbb{T}^r) \Rightarrow x = \bigcap \pi_k(\mathbb{T}^r_i) \ for \ each 2 \leq k \leq |\mathbb{T}^r|$$

.

The evaluation of transformation rule may produces an infinity of $\mathcal{M}$-Traces: all of them entail the substituted ungrouded patterns. Our interest is in one of them, the $\mathcal{M}$-Trace which represents exactly all observed elements, relations, attributs and values subsuming the ungrounded patterns without additionnal tuples. We can obtain such $\mathcal{M}$-Trace by intersection of entailed $\mathcal{M}$-Trace *w.r.t* of all its components.

Now that the rule transformation semantics has been specified, we can define an elementary transformation and its semantics.

Definition 17 (Elementary Transformation)**.** *A* elementary transformation *from an* $\mathcal{M}$*-Trace-model* $\mathcal{M}_{\mathbb{T}^r}$ *to an* $\mathcal{M}$*-Trace-model* $\mathcal{M}'_{\mathbb{T}^r}$ *is defined as a tuple*

$$\tau = (n, \mathcal{S}_{\mathcal{E}}, \mathcal{S}_\tau)$$

where

- *n is the name of the transformation,*
- $\mathcal{S}_\tau$ *is a not empty set of transformation rule.*

- $\mathcal{S}_\mathcal{E} = (\mathcal{AE}_b, \mathcal{AE}_e)$ *is a couple of arithmetic expression on* $\mathbb{T}^r$ *computing the bounds of temporal extension. Note that the* $\mathcal{S}_\mathcal{E}$ *may be empty.*

Definition 18 (Elementary Transformation Semantics)**.** *Let* $\mathbb{T}^r = (\mathcal{M}_{\mathbb{T}^r}, \mathcal{E}_\mathcal{T}, O, \lambda_C, \lambda_R, \lambda_A, \lambda_T)$*,* $\mathbb{T}^{r\prime} = (\mathcal{M}'_{\mathbb{T}^r}, \mathcal{E}'_\mathcal{T}, O', \lambda'_C, \lambda'_R, \lambda'_A, \lambda'_T)$ *be two* $\mathcal{M}$*-Traces. Let* $\tau = (n, \mathcal{S}_\mathcal{E}, \mathcal{S}_\tau)$ *be a elementary transformation. The* $\mathcal{M}$*-Trace resulting from evaluation of transformation* τ *according to* $\mathcal{M}'_{\mathbb{T}^r}$ *over* $\mathbb{T}^r$ *noted* $]\!]\tau[\![_{\mathbb{T}^r} = \bigcup \mathbb{T}^{r\prime}$*, where :*

- $\mathcal{M}'_{\mathbb{T}^r} = \mathcal{M}'_{\mathbb{T}^r}$
- $\pi_k(\mathbb{T}^{r\prime}) = \bigcup\limits_{s_i \in \mathcal{S}_\tau} \pi_k([\![s_i]\!])$ *for each* $3 \leq k \leq |\mathbb{T}^{r\prime}|$
- *if* $\mathcal{S}_\mathcal{E} = \varnothing$ *then* $\mathcal{E}_\mathcal{T} = [\min(b), \max(e)], \forall (o, [b, e]) \in \lambda_T$ *i.e.* $\mathcal{E}_\mathcal{T}$ *is the smallest temporal extension referencing all observed elements. Else if* $\mathcal{S}_\mathcal{E} = (\mathcal{AE}_b, \mathcal{AE}_e)$ *then* $\mathcal{E}_\mathcal{T} = (\alpha(\mathcal{AE}_b), \alpha(\mathcal{AE}_e))$ *where* α *is a the classical semantics function applied for arithmetic expressions.*

8 Queries and Transformations over an Online $\mathcal{M}$-Traces

The former formal model has fulfilled the needs of applications for complex off-line queries and transformations such as user-interactions analysis and extraction of users stored experiences. However, the requirements of most of applications do not fit the above description. One particularly interesting change is that $\mathcal{M}$-Trace may be exploited in real time, taking the form of an unbounded sequence of observed elements.

A defining characteristic of queries and transformations over on-line $\mathcal{M}$-Traces, is the potentially infinite and time-evolving nature of their inputs and outputs. New observed elements continually arrive on the input $\mathcal{M}$-Trace and new results are continually produced.

In this section, we summarize the following characteristics for the online $\mathcal{M}$-Traces and their processing requirements:

- An Online $\mathcal{M}$-Trace contains a potentially unbounded sequence of observed elements traced by an application.
- Observed elements arrive continuously at the system, pushed by the active tracing source. The Trace-based system neither has control over the order in which observed elements arrive nor over their arrival rates. Online observed elements rates and ordering could be unpredictable and vary over time.
- A collecting service transmits every observed element only once. As observed elements are accessed sequentially, an observed element that arrived in the past can be retrieved unless it is explicitly not stored.

- Queries and transformations over online $\mathcal{M}$-Traces are expected to run continuously and return new results as new observed elements arrive.

8.1 Online $\mathcal{M}$-Trace

Informally, we define $\mathcal{M}$-Trace as an append-only sequence of elements (at least timestamped and having an id) that arrive in some order. Since online $\mathcal{M}$-Trace elements may be tracked in bursts, observed elements may instead be modeled as a sequence with no order among elements that have arrived at the same time. We consider that each online observed item added to $\mathcal{M}$-Trace must have all its associated informations (timestamps and attribute values, identifier), that have arrived during the same unit of time. As Important consequence, the added observed element to $\mathcal{M}$-Trace at time t cannot be updated by adding values for its attributes in time $t' > t$. The only exception is when an observed element must be put in relation (one or plus) with others (one or plus) added before.

Definition 19 (Online $\mathcal{M}$-Trace)**.** *An Online $\mathcal{M}$-Trace is the tuple*

$$\mathbb{T}^s = (\mathcal{M}_{\mathbb{T}^s}, \hat{\mathcal{E}}_{\mathcal{T}}, O, \lambda_C, \lambda_R, \lambda_A, \lambda_T, \lambda_v)$$

consisting of:

- *a trace model $\mathcal{M}_{\mathbb{T}^r} = (\mathcal{T}, C, R, A, \text{dom}_R, \text{range}_R, \text{dom}_A, \text{range}_A)$ as defined above (definition 3),*
- *a right-unbounded[6] temporal extension $\hat{\mathcal{E}}_{\mathcal{T}}$,*
- *a potentially unbounded set O of observed elements,*
- *the functions id, λ_C, λ_A and a relation λ_R as specified in definition 4*
- *a total function $\lambda_T : O \rightarrow \mathbf{I}(\hat{\mathcal{E}}_{\mathcal{T}})$ called generating time labeling .*
- *A total function $\lambda_v : \hat{\mathcal{E}}_{\mathcal{T}} \rightarrow 2^O$ called arrival time labeling. At each instant $t \in \hat{\mathcal{E}}_T$, λ_v returns a finite subset from the set O of observed elements.*

As specified in definition 3, an online $\mathcal{M}$-Trace must be consistent to its model.

Note that two ordering of observed elements can be identified:

- Observed elements are ordered explicitly by their starting timestamps. This Order is provided by an application timestamp indicating their generation time (orders corresponding to $\leq_{\mathcal{T}}$).
- However, observed elements can arrive at a TBS out of order $\leq_{\mathcal{T}}$. Thus, the ordering of observed elements can also be defined implicitly by the arrival time at the trace-based system (order defined by $\leq_v$ according to λ_v).

[6] A time domain $\mathcal{T}$ is right-unbounded if it do not contains upper bound with respect to its order relationship. Formally, time domain $\mathcal{T}$ is right-unbounded if $\nexists\, t' \in \mathcal{T}$ such that $t \leq t'$ for all $t \in \mathcal{T}$.

In this paper, we do not make distinction between these two orders and we consider that $\leq_{\mathcal{T}}, \leq_v$ are order isomorphic[7]. If observed elements arrive at a TBS out of order and uncoordinated with each other, e.g. due to latencies introduced by a network, techniques from streaming research like the ones presented in [28] can be applied. Then we materialize this fundamental assumption by this constraint:

$$\forall o, o' \in O : o \leq_v o' \Leftrightarrow \lambda_v(o) \leq_{\mathcal{T}} \lambda_v(o') \Leftrightarrow \inf(\lambda_T(o)) \leq_{\mathcal{T}} \inf(\lambda_T(o'))$$

Note that the online $\mathcal{M}$-Trace may be regarded as $\mathcal{M}$-Trace evolving in time, so its current contents are all observed elements accumulated so far.

Definition 20 (Current online $\mathcal{M}$-Trace)**.** *Let* $\mathbb{T}^s = (\mathcal{M}_{\mathbb{T}^s}, \hat{\mathcal{E}}_{\mathcal{T}}, O, \lambda_C, \lambda_R, \lambda_A, \lambda_T, \lambda_v)$ *be an online* $\mathcal{M}$*-Trace. An online* $\mathcal{M}$*-Trace at time instant* $t \in \hat{\mathcal{E}}_T$*, noted* $\mathbb{T}^s(t)$ *is define as follow:* $\mathbb{T}^s(t) = \mathbb{T}^s$ *with* $O_t = \{o \in O \mid \lambda_v(o) \leq t\}$*.*

Also, we can define the resulting no-online $\mathcal{M}$*-Trace* $\mathbb{T}^r$ *at any time instant* $t \in \hat{\mathcal{E}}_T$ *as follow:* $\mathbb{T}^r$ *have a temporal extension* $\mathcal{E}_T$ *bounded by* t*, with the same elements* $O, \lambda_C, \lambda_R, \lambda_A$ *and* λ_T *than* $\mathbb{T}^s_t$ *and without* λ_v*.*

According to this definition, an evaluation of query Q at time t noted $[\![Q]\!]^t_{\mathbb{T}^s}$ must be equal to the result of a corresponding classical query evaluated on the current states of the online $\mathcal{M}$-Trace $\mathbb{T}^s(t)$ (equivalent to an $\mathcal{M}$-Trace $\mathbb{T}^r$).

We can define also the current observed elements of online $\mathcal{M}$-Trace at any distinct time instant t as a finite of observed elements with that specific arrival timestamp value.

Definition 21 (Current Observed Elements of online $\mathcal{M}$-Trace)**.** *Let* $\mathbb{T}^s = (\mathcal{M}_{\mathbb{T}^s}, \hat{\mathcal{E}}_{\mathcal{T}}, O, \lambda_C, \lambda_R, \lambda_A, \lambda_T, \lambda_v)$ *be an online* $\mathcal{M}$*-Trace. The current observed elements of online* $\mathcal{M}$*-Trace, noted* $\mathbb{T}^s_O(t) = \{o \in O \mid \lambda_v(o) = t\}$

8.2 Continuous Patterns and Queries over $\mathcal{M}$-Traces

Continuous queries are similar to conventional off-line queries, except that they are issued once and henceforth run *continually* over the online $\mathcal{M}$-Trace. As additions to the $\mathcal{M}$-Trace result in new query matches, the new results are returned to the user or application that issued the query. This section concentrates on the semantics of continuous queries.

Intuitively, the results of a continuous query on an online $\mathcal{M}$-Trace may be considered as a union of the result sets returned from successive query evaluations over the current online $\mathcal{M}$-Trace at every distinct time instant. Based on [33], we may formally define:

Definition 22 (Continuous Query over Online $\mathcal{M}$-Trace)**.** *Let* $Q = (n, \vartheta_{\mathcal{Q}}, P_{\mathcal{M}})$ *a continuous query submitted at time instant* $t_0 \in \mathcal{T}$ *on online* $\mathcal{M}$*-Trace* $\mathbb{T}^s$*. The results* $[\![Q]\!]^{t_i}_{\mathbb{T}^s}$ *that would be obtained at* $t_i \in \mathcal{T}$ *are the union of the*

[7] Two ordered sets $(A, \leq)$ and $(B, \leq)$ are order isomorphic iff there is a bijection f from A to B such that for all a_1,a_2 in A, $a_1 \leq a_2$ iff $f(a_1) \leq f(a_2)$.

subsets $[\![Q]\!]^{t}_{\mathbb{T}^s}$ *of substitutions from a series of queries* Q *on successive online* $\mathcal{M}$*-Traces* $\mathbb{T}^s(t)$*:*

$$\forall t_i \in \mathcal{T},\ t_i \geq t_0,\ [\![Q]\!]^{t_i}_{\mathbb{T}^s} = \bigcup_{t_0 \leq t \leq t_i} [\![Q]\!]^{t}_{\mathbb{T}^s}$$

The problem with this evaluation method is that it may not be practically feasible each time to compute query results by taking into account all online traces due to the overwhelming bulk of data that keep accumulating continuously.

That is, a solution suffices to re-evaluate the query over just the newly arrived elements and append qualifying substitutions to the result. Consequently, the answer of a query is a continuous append-only of results. Assuming that the extension temporal domain have an *unit* u, we define the incremental evaluation at time $t \in \mathcal{T}_u$ of continuous query Q over online $\mathcal{M}$-Trace $\mathbb{T}^s$ as follow:

Definition 23 (Incremental Evaluation of Continuous Query over Online $\mathcal{M}$-Trace)**.** *Let* $Q = (n, \vartheta_{\mathcal{Q}}, P_{\mathcal{M}})$ *a continuous query submitted at time instant* $t_0 \in \mathcal{T}_u$ *on online* $\mathcal{M}$*-Trace* $\mathbb{T}^s$*. The incremental evaluation* $[\![Q]\!]^{t_i}_{\mathbb{T}^s}$ *that would be obtained at* $t_i \in \mathcal{T}_u$ *is defined as follow:*

$$[\![Q]\!]^{t_i}_{\mathbb{T}^s} = \bigcup_{t=t_0+u}^{t_i} ([\![Q]\!]^{t}_{\mathbb{T}^s} - [\![Q]\!]^{t-u}_{\mathbb{T}^s}) \cup [\![Q]\!]^{t_0}_{\mathbb{T}^s}$$

This incremental evaluation is no better: If only intermediate stream contents are considered in each evaluation, it may happen that newer results may cancel observed elements (substitutions) included in formerly given answers.

A conservative approach is to accept queries with append-only results, thus not allowing any deletions or modifications at answers already produced. This class of continuous queries is called monotonic [33, 19]:

Definition 24 (Monotonic Continuous Query over Online $\mathcal{M}$-Trace)**.** *A* continuous *query* Q *applied over online* $\mathcal{M}$*-Trace* $\mathbb{T}^s$ *is characterized monotonic when*

$$\forall\ t_1, t_2 \in \mathcal{T}, t_1 \leq t_2, \mathbb{T}^s(t_1) \sqsubseteq \mathbb{T}^s(t_2) \Rightarrow [\![Q]\!]^{t_1}_{\mathbb{T}^s} \subseteq [\![Q]\!]^{t_2}_{\mathbb{T}^s}$$

where $\mathbb{T}^s_1 \sqsubseteq \mathbb{T}^s_2$ *means that* $\mathbb{T}^s_1$ *and* $\mathbb{T}^s_2$ *share the same trace model and* $\forall x \in \pi_i(\mathbb{T}^s_1) \Rightarrow x \in \pi_i(\mathbb{T}^s_2)$ *for each* $2 \leq i \leq |\mathbb{T}^s_1|$*.*

It is important to note that monotonicity refers to query results and not to incoming observed items. As long as elements may only be added to, but never discarded from results, incremental evaluation of basic queries involving selections patterns may be carried out as simple filters without particular complications. For example, querying a basic pattern[8] are monotonic over an online $\mathcal{M}$-Trace.

[8] simple conjunction of selection predicates on timestamps, id, attributes or relations of observed elements

Proposition 1. *Continuous querying of basic pattern over an online $\mathcal{M}$-Trace is monotonic.*

To see this, note that when a new observed element arrives, it either satisfies the basic pattern ($\mathbb{T}^s$ entails the grounded pattern P) or it does not and the satisfaction condition does not change over time. Consequently, the answer of a basic query is a continuous, append-only of results.

[33] was the first that defined a semantics for continuous queries over an append-only database. The semantics of monotonic queries was specified in term of incremental evaluation. In fact, [33] describes the class of continuous queries that can be rewritten and converted into incremental queries, i.e. can be evaluated periodically. Due to the problem of blocking operators as difference, the language is restricted to a subset of SQL. Law and al. identified in their later work [19] that the class of queries expressible by nonblocking operators corresponds to the class of monotonic queries.

Thus, in the context of general pattern, queries involving *without-opt* pattern are not monotonic since they use the *difference operator* in results. Intuitevely, querying genral pattern may produce results that cease to be valid as new observed element are added (difference may cancel previously results). In fact, it is well known that the negation is non-monotonic [19], even if issued over an append-only $\mathcal{M}$-Trace (e.g., "match from a $\mathcal{M}$-Trace of e-mail messages all those messages that have not yet received a reply").

It is obvious that continuous transformation of $\mathcal{M}$-Traces must deals with monotonic queries. Transforming non-monotonic queries over online $\mathcal{M}$-Traces are not possible if previously reported results can be removed if they cease to satisfy the pattern. In this case, we need to remove also observed elements generated by transformation rules referencing this pattern. Under the assumption of add-only observed elements in online $\mathcal{M}$-Traces, we can not transform and produce results that are valid at a given time and possibly invalidate them later. An example is shown in Figure **??**, where a observed elements O_2 was appended to the result because there did not exist any matching of (X *concerne* Y) at that time. However, the query results is empty after arrival of O_3 and thus, the transformation must delete a former generated element, violating the add-only assumption.

To define continuous transformation in incremental way, we need to considere only the case of monotonic pattern i.e. patterns .....

8.3 Continuous Transformations over $\mathcal{M}$-Traces

Incremental evaluation of transformation in our framework depends of incremental evaluation of its pattern. Let an elementary transformation $\tau = (n, \mathcal{S}_\tau)$ where $\mathcal{S}_\tau$ is a not empty set of transformation rule referencing continuous patterns. Let us consider the evaluation of τ over online $\mathcal{M}$-Trace $\mathbb{T}^s$ at time t as evaluation of $\mathcal{M}$-Trace $\mathbb{T}^r = \mathbb{T}^s(t)$ noted $[\![\tau]\!]^t_{\mathbb{T}^s} = \mathbb{T}^{s\prime}(t)$ where for each generated o in transformed online $\mathcal{M}$-Trace, the time arrival[9] $\lambda_v(o)$ is order isomorphic

[9] assigned by TBS

to starting time $\inf(\lambda_T(o))$. Under assumption of an extension temporal domain having an unit u, we define the incremental evaluation at time $t \in \mathcal{T}_u$ of continuous transformation τ over online $\mathcal{M}$-Trace $\mathbb{T}^s$ as follow:

Definition 25 (Incremental Evaluation of Continuous Transformation over Online $\mathcal{M}$-Trace)**.** *The incremental evaluation* $]\!]\tau[\![^{t_i}_{\mathbb{T}^s}$ *at time* $t_i \in \mathcal{T}_u$ *is defined as follow:*

$$]\!]\tau[\![^{t_i}_{\mathbb{T}^s} = \bigcup_{t=t_0+u}^{t_i} (]\!]\tau[\![^{t}_{\mathbb{T}^s} -]\!]\tau[\![^{t-u}_{\mathbb{T}^s}) \cup]\!]\tau[\![^{t_0}_{\mathbb{T}^s}$$

where $\mathbb{T}^s(t_1) - \mathbb{T}^s(t_2)$ *produce an online* $\mathcal{M}$*-Trace with* $O = O(t_1) - O(t_2)$

9 Implementation of $\mathcal{M}$-Trace Queries and Transformations

We define a translation from $\mathcal{M}$-Trace Queries and Transformations to Datalog which can serve straightforwardly to implement TBS core within existing rules engines. This translation allows us also to study expressivity and complexity of our language. We start by a brief description of datalog, then we define a translation for the $\mathcal{M}$-Trace patterns, which we will extend thereafter towards transformation.

9.1 Datalog

The implementation of a $\mathcal{M}$-Trace query and transformation is then defined using a translation to corresponding datalog program. Our mapping function is called m and defined in the following[10]. We use the following syntax for datalog:

- A datalog program is a set of normal clauses.
- A normal clause has the form $L :- L_i$. L is called the head and L_i the body of the clause. L is an atom and L_i is a conjunction of literals.
- "," in the body of a clause stands for conjunction. $\neg$ and $\vee$ are used for negation and disjunction.

In this report we will use a very general form of Datalog commonly referred to as Answer Set Programming (ASP), i.e. function-free logic programming (LP) under the answer set semantics [11]. ASP is widely proposed as a useful tool for various problem solving tasks in e.g. Knowledge Representation and Deductive databases. ASP extends Datalog with useful features such as negation as failure, disjunction in rule heads, etc.(For Datalog details see Appendix I). In the following, we will assume that constants are quoted '“' conversely to variables which are not. Thus, the constant v_1 is denoted by 'v_1' and variable x_1 is denoted only by x_1.

[10] We will use functional and relational syntax interchangeably where userful, i.e. for $f(x) = y$ we will write $(x, y) \in f$ and analogous for the inverse: $f^{-1}(y) = x$ and $(y, x) \in f^{-1}$

9.2 Expressive Power of Languages

By the expressive power of a query language, we understand the set of all queries expressible in that language [1]. In order to determine the expressive power of a query language L, usually one chooses a well-studied query language L' and compares L and L' in their expressive power. Two query languages have the same expressive power if they express exactly the same set of queries.

A given query language is defined by a triple $(\mathbb{Q}, \mathbb{D}, \mathbb{S}, [\![.]\!])$, where $\mathbb{Q}$ is a set of queries, $\mathbb{D}$ is a set of databases, $\mathbb{S}$ is a set of solutions, and $[\![.]\!] : \mathbb{Q} \times \mathbb{D} \rightarrow \mathbb{S}$ is the evaluation function. The evaluation of a query $Q \in \mathbb{Q}$ on a database $D \in \mathbb{D}$ is denoted $[\![Q]\!]_D$.

Considering a language $L = (\mathbb{Q}, \mathbb{D}, \mathbb{S}, [\![.]\!])$, two queries Q_1,Q_2 of L are equivalent, denoted $Q_1 \equiv Q_2$, if they return the same answer for all input databases, i.e., $[\![Q_1]\!]_D = [\![Q_2]\!]_D$ for every $D \in \mathbb{D}$. Let L_1, L_2 be two fragments of L. We say that L_1 is contained in L_2 , if and only if for every query Q_1 in L_1 there exists a query Q_2 in L_2 such that $Q_1 \equiv Q_2$. To compare two query languages with different syntax and semantics require having a common data and language setting to do the comparison. Let $L_1 = (\mathbb{Q}_1, \mathbb{D}_1, \mathbb{S}_1, [\![.]\!]^1)$ and $L_2 = (\mathbb{Q}_2, \mathbb{D}_2, \mathbb{S}_2, [\![.]\!]^2)$be two query languages. We now say that L_1 is contained in L_2 if and only if there are bijective translations $\mathfrak{m}_D : \mathbb{D}_1 \rightarrow \mathbb{D}_2$, and $\mathfrak{m}_S : \mathbb{S}_1 \rightarrow \mathbb{S}_2$ and query translation $\mathfrak{m}_Q : \mathbb{Q}_1 \rightarrow \mathbb{Q}_2$, such that for all $Q_1 \in \mathbb{Q}_1$ and $D_1 \in \mathbb{D}_1$ it holds

$$\mathfrak{m}_S([\![Q_1]\!]^1_D) = [\![\mathfrak{m}_Q(Q_1)]\!]^2_{\mathfrak{m}_D(D_1)}.$$

9.3 From $\mathcal{M}$-Trace Queries to Datalog

Informally, Datalog facts correspond to observed elements associated with thier timestamps, relations and attribut values. Datalog rules correspond to $\mathcal{M}$-Trace patterns, goal queries correspond to query variable, and the set of substitutions returned by a Datalog query corresponds to the set of substitutions returned by a $\mathcal{M}$-Trace query.

Note that because $\mathcal{M}$-Trace Queries and Datalog programs have different type of input and output formats, we have to normalize them to be able to do the translation. The general idea of the translations is the following.

- A fact OBSERVED(‘$\mathbb{T}^r_1$‘, ‘o_1‘) models an observed element o_1 occuring in the $\mathcal{M}$-Trace named $\mathbb{T}^r_1$.
- A fact OBSERVEDTYPE(‘$\mathbb{T}^r_1$‘, ‘o_1‘ , ‘c_1‘) models an observed element o_1 of type c_1 which occurs in the $\mathcal{M}$-Trace named $\mathbb{T}^r_1$.
- A fact OBSERVEDRELATION(‘$\mathbb{T}^r_1$‘, ‘r_1‘, ‘o_1‘ , ‘o_2‘) models a relation r_1 between two observed elements o_1, o_2 which occurs in the $\mathcal{M}$-Trace named $\mathbb{T}^r_1$.
- A fact OBSERVEDATTRIBUTE(‘$\mathbb{T}^r_1$‘, ‘o_1‘ , ‘a_1‘, ‘v_1‘) models an observed element o_1 having an attribute a_1 with the value v_1 which occurs in the $\mathcal{M}$-Trace named $\mathbb{T}^r_1$. The same fact can will be used to define both observed

element time attributes (*.begin, .end and .arrival*) and observed elements *.id* attribute.

- The unary predicate NULL encodes the *'null'* value.

With respect to solutions, note that naturally sets of substitutions in $\mathcal{M}$-Trace queries correspond bijectively to sets of substitutions in Datalog.

Let $L_{\mathbb{T}^r} = (\mathbb{Q}_{\mathbb{T}^r}, \mathbb{D}_{\mathbb{T}^r}, \mathbb{S}_{\mathbb{T}^r}, [\![.]\!]_{\mathbb{T}^r})$ and $L_d = (\mathbb{Q}_d, \mathbb{D}_d, \mathbb{S}_d, ans_d)$ be the $\mathcal{M}$-Trace language and the Datalog language respectively. To map $L_{\mathbb{T}^r}$ to L_d, we define translations $\mathfrak{m}_Q : \mathbb{Q}_{\mathbb{T}^r} \rightarrow \mathbb{Q}_d$, $\mathfrak{m}_D : \mathbb{D}_{\mathbb{T}^r} \rightarrow \mathbb{D}_d$, and $\mathfrak{m}_S : \mathbb{S}_{\mathbb{T}^r} \rightarrow \mathbb{S}_d$. That is, $\mathfrak{m}_Q$ translates a $\mathcal{M}$-Trace query into a Datalog query, $\mathfrak{m}_D$ translates a $\mathcal{M}$-Trace dataset into a set of Datalog facts, and $\mathfrak{m}_S$ translates a set of $\mathcal{M}$-Trace solution mappings into a set of Datalog substitutions.

However, we can distinguish two approaches of converting ontology-like models into logic programs: the direct mapping approach described in [15] and meta mapping approach [35]. The former have some significant scalability deficits as well as representational drawbacks (see [35] for more details). Therefore, we have chosen a meta mapping approach for the main reason that this translation is especially suitable for storing and processing a large $\mathcal{M}$-Traces within logical databases. The following part presents algorithms defining our mapping.

9.3.1 $\mathcal{M}$-Trace as Datalog Knowledge Base

Let $\mathbb{T}^r$ be an $\mathcal{M}$-Trace associated to model $\mathcal{M}_{\mathbb{T}^r}$. We denote by $\mathfrak{m}_{\mathcal{M}_{\mathbb{T}^r}}(\mathcal{M}_{\mathbb{T}^r})$ the function which translates a trace model $\mathcal{M}_{\mathbb{T}^r}$ into a set of Datalog facts and rules. The function $\mathfrak{m}_{\mathcal{M}_{\mathbb{T}^r}}$ is defined by the following algorithm:

Algorithm 1: Translation function definition $\mathfrak{m}_{\mathcal{M}_{\mathbb{T}^r}}$ from $\mathcal{M}_{\mathbb{T}^r}$ to Datalog

Input : an trace model $\mathcal{M}_{\mathbb{T}^r}$ identified by $'\mathcal{M}'_{\mathbb{T}^r}$
Output: a set of Datalog facts F and rules R

begin
- $F \leftarrow \emptyset$
- $R \leftarrow \emptyset$
- Add the Datalog fact NULL(‘null‘)
- **foreach** $c \in C$ *in* $\mathcal{M}_{\mathbb{T}^r}$ **do**
  - Add the fact TYPE(‘$\mathcal{M}_{\mathbb{T}^r}$‘,‘$c$‘) to F
- **foreach** $r \in R$ *in* $\mathcal{M}_{\mathbb{T}^r}$ **do**
  - Add the fact RELATIONTYPE(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘$r$‘, ‘$c_i$‘, ‘$c_j$‘)
- **foreach** $(c_i, c_j) \in \ \leq_C$ *in* $\mathcal{M}_{\mathbb{T}^r}$ **do**
  - Add the fact SUBTYPE(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘$c_i$‘, ‘$c_j$‘) to F
- **foreach** $(r_i, r_j) \in \ \leq_R$ *in* $\mathcal{M}_{\mathbb{T}^r}$ **do**
  - Add the fact SUBRELATIONTYPE(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘$r_i$‘, ‘$r_j$‘) to F
- **foreach** $(r_i, c_i) \in \text{dom}_R$ *in* $\mathcal{M}_{\mathbb{T}^r}$ **do**
  - Add the fact DOMAIN(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘$r_i$‘, ‘$c_i$‘) to F
- **foreach** $(r_i, c_i) \in \text{range}_R$ *in* $\mathcal{M}_{\mathbb{T}^r}$ **do**
  - Add the fact RANGE(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘$r_i$‘, ‘$c_i$‘) to F
- **foreach** $(a_i, c_i) \in \text{dom}_A$ *in* $\mathcal{M}_{\mathbb{T}^r}$ **do**
  - Add the fact DOMAIN(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘$a_i$‘, ‘$c_i$‘) to F
- **foreach** $(a_i, d_i) \in \text{range}_A$ *in* $\mathcal{M}_{\mathbb{T}^r}$ **do**
  - Add the fact RANGE(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘$a_i$‘, ‘$d_i$‘) to F
- **if** $\mathcal{T}$ *is* $\mathcal{T}_{seq}$ **then**
  - Add the fact TEMPORALDOMAIN(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘*sequential*‘) to F
- **if** $\mathcal{T}$ *is* $\mathcal{T}_{unit}$ **then**
  - Add the fact TEMPORALDOMAIN(‘$\mathcal{M}_{\mathbb{T}^r}$‘, ‘*unit*‘) to F
- Add the following rules to R
  - (1) SUBTYPE($\mathcal{M}_{\mathbb{T}^r}, c, c'$) :- SUBTYPE($\mathcal{M}_{\mathbb{T}^r}, c, c''$) , SUBTYPE($\mathcal{M}_{\mathbb{T}^r}, c'', c'$)
  - (2) SUBRELATIONTYPE($\mathcal{M}_{\mathbb{T}^r}, r, r'$) :- SUBRELATIONTYPE($\mathcal{M}_{\mathbb{T}^r}, r, r''$) , SUBRELATIONTYPE($\mathcal{M}_{\mathbb{T}^r}, r'', r'$)

end
return *F,R*

Basically, The algorithm 1 convert the trace model from $\mathcal{M}$-Trace into a set of facts reflecting the content of the ontology describing the $\mathcal{M}$-Trace. In order to reflect the underlying semantic of the introduced trace model, we will have to add some rules, which work on the given facts. As the rule can be used with any combination of bound and free variables, every kind of type-instance query is possible. Additionally the transitivity of the subrelation and subtype relationship is covered by the rules (1,2).

The following algorithm 2 gives the complete definition of the required rules and facts to translate an $\mathcal{M}$-Trace.

Algorithm 2: Translation function $\mathfrak{m}_D$ from $\mathcal{M}$-Trace to Datalog

Input : an $\mathcal{M}$-Trace $\mathbb{T}^r$ (or $\mathbb{T}^s$) identified by $'\mathbb{T}^r'$ (or $'\mathbb{T}^s'$)
Output: a set of Datalog facts F and rules R

begin
- $(F, R) = \mathfrak{m}_{\mathcal{M}_{\mathbb{T}^r}}(\mathcal{M}_{\mathbb{T}^r})$
- **foreach** $(o_i, id_i) \in id$ *in* $\mathbb{T}^r$ **do**
  - Add the fact observedAttribute('$\mathbb{T}^r$', 'o_i', 'id_i') to F
- **foreach** $(o_i, c_i) \in \lambda_C$ *in* $\mathbb{T}^r$ **do**
  - Add the fact observedType('$\mathbb{T}^r$', 'o_i', 'c_i') to F
- **foreach** $(o_i, o_j, r_i) \in \lambda_R$ *in* $\mathbb{T}^r$ **do**
  - Add the fact observedRelation('$\mathbb{T}^r$', 'r_i', 'o_i', 'o_j') to F
- **foreach** $(o_i, a_j, v_i) \in \lambda_A$ *in* $\mathbb{T}^r$ **do**
  - Add the fact observedAttribute('$\mathbb{T}^r$', 'o_i', 'a_i', 'v_i') to F
- **foreach** $(o_i, [b_i, e_i]) \in \lambda_T$ *in* $\mathbb{T}^r$ **do**
  - Add the fact observedAttribute('$\mathbb{T}^r$', 'o_i', '*begin*', 'b_i') to F Add the fact observedAttribute('$\mathbb{T}^r$', 'o_i', '*end*', 'e_i') to F
- **if** *the input is an off-line* $\mathcal{M}$*-Trace* $\mathbb{T}^r$ **then**
  - Add the fact temporalExtension('$\mathbb{T}^r$', $\inf(\mathcal{E}_\mathcal{T})$, $\sup(\mathcal{E}_\mathcal{T})$ to F
- **if** *the input is an on-line* $\mathcal{M}$*-Trace* $\mathbb{T}^s$ **then**
  - Add the fact temporalExtension('$\mathbb{T}^s$', $\inf(\mathcal{E}_\mathcal{T})$, null) to F
  - **foreach** $(o_i, v_i) \in \lambda_T$ *in* $\mathbb{T}^s$ **do**
    - Add the fact observedAttribute('$\mathbb{T}^s$', 'o_i', '$arrival'$, 'v_i') to F
- Add the following rules to R
  - (1) observedType($\mathbb{T}^r, o, c$) :-observedType($\mathbb{T}^r, o, c'$) , subType($\mathcal{M}_{\mathbb{T}^r}, c', c$)
  - (2) observedRelation($\mathbb{T}^r, r, o, o'$) :- observedRelation($\mathbb{T}^r, r', o, o'$) , subRelationType($\mathcal{M}_{\mathbb{T}^r}, r', r$)
  - (3) observed($\mathbb{T}^r, o$) :- observedType($\mathbb{T}^r, o, c$)
  - (5) observed($\mathbb{T}^r, o$) :- observedAttribute($\mathbb{T}^r, o, a, v$)
  - (6) observed($\mathbb{T}^r, o$), observed($\mathbb{T}^r, o'$) :- observedRelation($\mathbb{T}^r, r, o, o'$)
- **return** F,R

end

The specified rule (1) defines that if an obseved o is instance of type c' and c' is subtype of type c, o is also an instance of type c. As you can see the above rules are completely independent of any entities defined in the $\mathcal{M}$-Trace and can thus be used for every $\mathcal{M}$-Trace. With the combination of the $\mathcal{M}_{\mathbb{T}^r}$ specific facts and the general rule we can now perform all $\mathcal{M}$-Trace queries.

9.3.2 $\mathcal{M}$-Trace patterns as Datalog rules

Let P be a $\mathcal{M}$-Trace pattern to be evaluated against an $\mathcal{M}$-Trace $\mathbb{T}^r$. We denote by $\mathfrak{m}_P(P, \mathbb{T}^r)$ the function which translates the $\mathcal{M}$-Trace pattern P into a set of Datalog facts. The complete rules work as follows. We denote by $\overline{\vartheta(P)}$, a tuple of variables obtained from a lexicographical ordering of the variables in the $\mathcal{M}$-Trace pattern P. Then, the function $\mathfrak{m}_p(P, \mathbb{T}^r)$ is defined recursively in algorithm 3. A predicate comp implements the notion of compatible substitutions: comp(o, o) :- observed($\mathbb{T}^r, o$); comp(o, '*null*') :- observed($\mathbb{T}^r, o$) and comp('*null*', o) :- observed($\mathbb{T}^r, o$).

Algorithm 3: $\mathcal{M}$-Trace pattern translation function $\mathfrak{m}_P$

Input : an $\mathcal{M}$-Trace $\mathbb{T}^r$ pattern P
Output: a set of Datalog rules

begin

if P *is* $(X \ : \ C_i)$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \{\text{SUBSTITUTION}_i(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P)}) \text{ :- } \text{OBSERVEDTYPE}(\text{`}\mathbb{T}^r\text{`}, X, \text{`}C_i\text{`})\}$

if P *is* $(X \ r_i \ Y)$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \{\text{SUBSTITUTION}_i(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P)}) \text{ :-}$
$\text{OBSERVEDRELATION}(\text{`}\mathbb{T}^r\text{`}, \text{`}r_i\text{`}, X, Y)\}$

if P *is* $X.a_i \diamondsuit Y.a_i)$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \{\text{SUBSTITUTION}_i(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P)}) \text{ :-}$
$\text{OBSERVEDATTRIBUTE}(\text{`}\mathbb{T}^r\text{`}, X, \text{`}a_i\text{`}, Z_1),$
$\text{OBSERVEDATTRIBUTE}(\text{`}\mathbb{T}^r\text{`}, Y, \text{`}a_i\text{`}, Z_2),$
$Z_1 \diamondsuit Z_2 \ \}$
where a_i *can be a defined attribute or begin, end or id attribute;*
and $\diamondsuit \in \{=, \neq, <, \leq, \geq, >\}$

if P *is* $(X.a_i \diamondsuit v)$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \{\text{SUBSTITUTION}_i(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P)}) \text{ :-}$
$\text{OBSERVEDATTRIBUTE}(\text{`}\mathbb{T}^r\text{`}, X, \text{`}*\text{`}, Z),$
$Z \diamondsuit v\}$
where a_i *can be a defined attribute or begin, end or id attribute;*
and $\diamondsuit \in \{=, \neq, <, \leq, \geq, >\}$

if P *is* $q(x_1, ..., x_n))$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \mathfrak{m}_\mathcal{Q}(q, i+1) \cup \{\text{SUBSTITUTION}_i(\overline{x_1, ..., x_n}) \text{ :- } q(\overline{x_1, ..., x_n})\}$
where $\mathfrak{m}_\mathcal{Q}$ *is* $\mathcal{M}$*-Trace query translation (see algorithm 4)*

if P *is* $(P' \ , \ P'')$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \mathfrak{m}_P(P', \mathbb{T}^r, 2*i) \cup \mathfrak{m}_P(P'', \mathbb{T}^r, 2*i+1) \cup$
$\{\text{SUBSTITUTION}_i(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P)}) \text{ :- } \text{SUBSTITUTION}_{2*i}(\text{`}\mathbb{T}^r\text{`}, \overline{\nu_1(\vartheta(P'))}),$
$\text{SUBSTITUTION}_{2*i+1}(\text{`}\mathbb{T}^r\text{`}, \overline{\nu_2(\vartheta(P''))}),$
$\text{COMP}(\nu_1(x_1), \nu_2(x_1)),...,\text{COMP}(\nu_1(x_n), \nu_2(x_n))\}$
where $x_{i(i=1..n)} \in \vartheta(P') \ \cap \ \vartheta(P'')$
and $\nu_i : \vartheta \to \vartheta$ *a variable-renaming function*

if P *is* $(P' \ or \ P'')$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \mathfrak{m}_P(P', \mathbb{T}^r, 2*i) \ \cup \ \mathfrak{m}_P(P'', \mathbb{T}^r, 2*i+1) \cup$
$\{\text{SUBSTITUTION}_i(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P)}) \text{ :- } \text{SUBSTITUTION}_{2*i}(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P')}),$
$\text{NULL}(x''_1),\text{NULL}(x''_2),...,\text{NULL}(x''_n)\} \cup$
$\{\text{SUBSTITUTION}_i(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P)}) \text{ :- } \text{SUBSTITUTION}_{2*i+1}(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P'')}),$
$\text{NULL}(x'_1),\text{NULL}(x'_2),...,\text{NULL}(x'_n)\}$
where $x'_{i(i=1..n)} \in \vartheta(P) \setminus \vartheta(P'')$
and $x''_{i(i=1..n)} \in \vartheta(P) \setminus \vartheta(P')$

if P *is* $(P' \ without \ P'')$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \mathfrak{m}_P(P', \mathbb{T}^r, 2*i) \ \cup \ \mathfrak{m}_P(P'', \mathbb{T}^r, 2*i+1) \cup$
$\{\text{SUBSTITUTION}_i(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P)}) \text{ :- } \text{SUBSTITUTION}_{2*i}(\text{`}\mathbb{T}^r\text{`}, \overline{\vartheta(P')}),$
$\text{NOT COMP}(x'_1, x''_1), \text{ NOT COMP}(x'_1, x''_2),..., \text{ NOT COMP}(x'_1, x''_m)$
$\text{NOT COMP}(x'_2, x''_1),\text{NOT COMP}(x'_2, x''_2),..., \text{ NOT COMP}(x'_2, x''_m)$
...
$\text{NOT COMP}(x'_n, x''_1),\text{NOT COMP}(x'_n, x''_2),..., \text{ NOT COMP}(x'_n, x''_m)\}$
where $x'_{i(i=1..n)} \in \vartheta(P')$
and $x''_{j(i=1..m)} \in \vartheta(P'')$

if P *is* $(P' \ opt \ P'')$ **then**

$\mathfrak{m}_P(P, \mathbb{T}^r, i) = \mathfrak{m}_P((P', P''), \mathbb{T}^r, i) \ \cup \ \mathfrak{m}_P((P' \ without \ P''), \mathbb{T}^r, i)$

end

To define a translation for $\mathcal{M}$-Trace pattern results joins involving *null* values, we define ν_i a variable-renaming function. Obviously, since we allow null-bindings to join with any other value, $\mathcal{M}$-Trace pattern results joining semantics is tricky to achieve. For that, we have define $\nu_i : \vartheta \rightarrow \vartheta$ a variable-renaming function, with ν_i satisfies that $\text{dom}(\nu_1) = \text{dom}(\nu_2) = \vartheta(P') \cap \vartheta(P'')$ and $\text{range}(\nu_1) \cap \text{range}(\nu_2) = \emptyset$. Our idea is to rename each variable $v_i \in \vartheta(P') \cap \vartheta(P'')$ in the respective rule bodies to v_i' or v_i'', respectively, in order to disambiguate the occurrences originally from sub-pattern P' or P'', respectively.

9.3.3 $\mathcal{M}$-Trace queries as Datalog queries

Let $\mathcal{Q}_{\mathbb{T}^r} = (n_\mathcal{Q}, \vartheta_\mathcal{Q}, P_\mathcal{M})$ be a query and $\mathbb{T}^r \in \mathbb{D}_{\mathbb{T}^r}$ an $\mathcal{M}$-Trace. The function $\mathfrak{m}_Q(\mathcal{Q}_{\mathbb{T}^r}, i)$ returns the Datalog query $\mathcal{Q}_d = (\Pi, n_\mathcal{Q}(\overline{\vartheta(\mathcal{Q})})$ where $\overline{\vartheta(\mathcal{Q})}$ is a tuple of variables obtained from a lexicographically ordering of the variables in $\vartheta_\mathcal{Q}$, $n_\mathcal{Q}$ a predicat having the same name of query and Π is the Datalog program defined as follow:

Algorithm 4: $\mathcal{M}$-Trace query translation function $\mathfrak{m}_\mathcal{Q}$

Input : an $\mathcal{M}$-Trace $\mathbb{T}^r$ identified by '$\mathbb{T}^r$', a query $\mathcal{Q}_{\mathbb{T}^r}$ named $n_\mathcal{Q}$ and an indice i
Output: a Datalog program Π

begin
  $\Pi \leftarrow \emptyset$
  if *i is not defined* **then**
    $\Pi \leftarrow \{n_\mathcal{Q}(\text{'}\mathbb{T}^r\text{'}, \overline{\vartheta(\mathcal{Q})}) \text{ :- } \text{SUBSTITUTION}_1(\text{'}\mathbb{T}^r\text{'}, \overline{\vartheta(\mathcal{Q})})\} \cup \mathfrak{m}_P(P_\mathcal{M}, \mathbb{T}^r, 1)$
    This is the case of classic query evaluation
  else
    $\Pi \leftarrow \{n_\mathcal{Q}(\text{'}\mathbb{T}^r\text{'}, \overline{\vartheta(\mathcal{Q})}) \text{ :- } \text{SUBSTITUTION}_i(\text{'}\mathbb{T}^r\text{'}, \overline{\vartheta(\mathcal{Q})})\} \cup \mathfrak{m}_P(P_\mathcal{M}, \mathbb{T}^r, i)$
    This is the case of nested query evaluation embeded in pattern $q(x_i)$
end
return Π

The first rule serves to evaluate a query on $\mathcal{M}$-Trace. The second rule allows the add rules in the context of pattern involving a call of query. Naturally, the resulting programs possibly involve recursion, and, even worse, recursion over negation as failure. Fortunately, the general answer set semantics, which we use, can cope with this. A more in-depth investigation of the complexity and other semantic features of such a combination is on our agenda.

9.3.4 $\mathcal{M}$-Trace queries solutions as Datalog solutions

We obtained by translating an $\mathcal{M}$-Trace $\mathcal{Q}_{\mathbb{T}^r}$ a datalog query $\mathfrak{m}_\mathcal{Q} = \mathcal{Q}_d = (\Pi, n_\mathcal{Q}(\overline{\vartheta(\mathcal{Q})}))$. The evaluation of $\mathcal{Q}_d$ denoted $ans_d(Q) = \{\theta \mid \theta(L) \in \text{facts}^*(\Pi)\}$. Given a set of substitutions Θ, the set of mappings obtained from Θ, denoted $\mathfrak{m}_S(\Theta)$, is defined as follows: for each substitution $\theta \in \Theta$ there exists a mapping $\psi \in \mathfrak{m}_S(\Theta)$ satisfying that, if $x/t \in \theta$ then $x \in \text{dom}(\psi)$ and $\psi(x) = t$[11].

[11] Due to the similarity of the objects and to avoid complicating the notation, we will not distinguish between substitutions and mappings. That is, will consider $\mathfrak{m}_S$ as the identity. Formally, given a result set φ, the set of substitutions obtained from φ, denoted $\mathfrak{m}_S(\varphi)$, is defined as follows: for each substitution $\psi \in \varphi$ there exists a substitution $\theta \in \mathfrak{m}_S(\varphi)$ satisfying

9.4 From $\mathcal{M}$-Trace Transformations to Datalog

We have covered only $\mathcal{M}$-Trace queries so far. The transformation of pattern result form, which allows to construct new observed elements could be emulated in Datalog as well. Namely, we can allow transformations of the form $\tau = (n, \mathcal{S}_\tau)$ where $\mathcal{S}_\tau$ is a set of transformation rule $(P_\mathcal{M}, G)$ consisting of model patterns $P_\mathcal{M}$ and template G. We can model such transformations by adding the datalog rules defined by algorithm 5.

Algorithm 5: $\mathcal{M}$-Trace Transformations translation function $\mathfrak{m}_\tau$

Input : $\mathcal{M}$-Traces $\mathbb{T}^r, \mathbb{T}^{r\prime}$ identified by '$\mathbb{T}^r$', '$\mathbb{T}^{r\prime}$', a transformations $\tau = (n, \mathcal{S}_\tau)$
Output: a Datalog program Π

begin
  $\Pi \leftarrow \emptyset$
  add $\mathbb{T}^r$ to Π using $\mathfrak{m}_D$
  add $\mathcal{M}'_{\mathbb{T}^r}$ to Π using $\mathfrak{m}_{\mathcal{M}_{\mathbb{T}^r}}$
  foreach $(P_\mathcal{M}, G) \in \mathcal{S}_\tau$ **do**
    $\Pi \leftarrow \Pi \cup \mathfrak{m}_P(P_\mathcal{M})$
    foreach $g_i \in G$ **do**
      if g_i *is* $(w : c)$ **then**
        Add to Π the rule
        observedType('$\mathbb{T}^{r\prime}$', w, 'c') :- substitution$_1$('$\mathbb{T}^{r\prime}$', w)
      if g_i *is* $(w\ r\ w')$ **then**
        Add to Π the rule
        observedRelation('$\mathbb{T}^{r\prime}$', r, $\overline{w, w'}$) :- substitution$_1$('$\mathbb{T}^{r\prime}$', $\overline{w, w'}$)
      if g_i *is* $(w.a = v)$ **then**
        Add to Π the rule
        observedAttribute('$\mathbb{T}^{r\prime}$', w, 'a', 'v') :- substitution$_1$('$\mathbb{T}^{r\prime}$', w)
end
return Π

The result $\mathcal{M}$-Trace is then naturally represented in the answer set of the program extended that way in the extension of the predicate observed('$\mathbb{T}^{r\prime}$', x).

10 Related Work

In this section we provide a general discussion of past work that relates in some way to our work.

Even if the notion of trace plays an important role in many of applications and research fields (e.g. user modeling Acquisition [26, 17, 30], web-based logs and click streams [16, 9, 5]), to the best of our knowledge, there is neither explicit theorisation of this notion as a specific object having interesting properties nor generic view of what is a trace. Usually, most of techniques performed over traces in these contexts aim to exploit machine learning and statistical tools for mining and extracting the relevant knowledge (e.g. Web Usage Mining [22, 4, 23, 24, 20]). Even if these techniques deal efficiently with the need to process traces consistent with simple models, extending them towards ontologies

that, for each $x \in \text{dom}(\psi)$ there exists $x/t \in \theta$ such that $t = \psi(x)$ when $\psi(x)$ is bounded and $t = null$ otherwise.

involving hierarchies is often non-trivial. Besides, storing and querying trace-like sources often relies on relational database than ontological repositories. Thus, the exploitation and processing of traces draws its formal bases on theories related to well-studied database model and its extensions([1, 32, 27, 10, 21]). However, exploiting the relational algebra or a temporal algebra over $\mathcal{M}$-Traces is non-trivial for a number of reasons, including the following. $\mathcal{M}$-Traces involve ontologies: It is not apparent how the stored and instantiated ontology can be issued and transformed by SQL-like languages. Although, we can use a real ontology-querying language like SPARQL [8, 6], but still to define what is the semantics of such language in the case of online $\mathcal{M}$-Traces and continuous queries and transformations. Query and transformation formulation only makes sense if the semantics is precisely defined. To guarantee predictable and repeatable query and transformation results, continuous query and transformation semantics should be defined independent of how the system operates internally.

In fact, the fundamental differences between queries over online and offline data is not novel and has been widely identified by Stream Management community[12]. The defined stream algebra is often reducible to its relational analogue [18]. This important property ensures semantic compliance and allows systems to carry over the algebraic equivalences from the extended relational algebra to stream algebra. Despite similarities and bridges between algebra and languages, one-time and continuous queries rely often on different semantics. As we are motivated by querying $\mathcal{M}$-Traces both in online and offline situation under the same semantics, most of stream languages seem to be tricky to extend specially under ontological model.

In addition, even if querying paradigm for streaming data distinguish several declarative languages having SQL-like syntax and stream-specific semantics, as CQL [2], a few of them employ a model supporting type-hierarchies (see Tribeca [31] and COUGAR [36]). Beyond this, in most of stream algebra, it is not evident how several and heterogeneous notion of time involved in $\mathcal{M}$-Traces are supported. For instance, it is not apparent how timestamps are assigned to the operator results. Assume we want to compute a join over two $\mathcal{M}$-Trace patterns whose elements are integer-timestamp pairs. Is a join result tagged with the minimum or maximum timestamp of the qualifying elements, or both? What happens in the case of cascading joins? While some approaches in stream management prefer to choose a single timestamp, e.g.,[12], others suggest keeping all timestamps to preserve the full information [14, 18]. To the best of our knowledge, currently there is no exists stream framework dealing with heterogeneous time and temporal domain descriptions.

Beyond this, the last relevant reason differentiating $\mathcal{M}$-Traces from stream is the transformation operator. If stream operator allows to transform an input stream S_1 to an output S_2, this transformation keeps the same schema for S_1 and S_2 ($S_2 \subseteq S_1$). Thus, a transformation in this case is the result of patterns expressed in query and matched in the input. In our framework, the

[12]The assumption in this field is that a Database Management System (DBMS) handles transient queries over persistent data, whereas a Data Stream Management System (DSMS) processes persistent queries over transient data [13].

transformations are not the result of query, but they operates on query results by using rules allowing to express a complicated processing. Since the input $\mathcal{M}$-Trace and output are modeled with different ontologies, such transformations are difficult to express in most of stream management system.

In sum, transforming and querying $\mathcal{M}$-Traces relies and combines several theoretical model. To deal with aimed properties underlying trace exploitations, we define a common semantics incorporating inherently the several notion of time specific to $\mathcal{M}$-Traces. Such framework allows a user or artificial agent to exploit the maded traces during or afterwards observation of his/her activity within unified framework. However, as it seems to be a good idea to formalize the notion of $\mathcal{M}$-Trace with the associated languages in common framework, we will have to face to several choices and problems that we discuss in the next section.

11 Discussion and Open Issues

Unlike traditional one-time queries, continuous queries produce output over time. Due to fact that queries are long-running and online $\mathcal{M}$-Trace can be unbounded, a no-incremental evaluation methods are not suitable for $\mathcal{M}$-Trace processing. However, incremental evaluation involves monotonic queries which can be re-executed over a part of trace newly observed and increment without delete the last evaluation.

Thus, dealing with non-monotonic queries goes against the incremental evaluation approach. In fact, the problem of the non-monotonic queries has long been recognized by data stream researchers, who have proposed the use of devices such as punctuation [34] and windows [3] to address this problem. While these approaches deal effectively with important aspects of the problem, they do not solve the problem of identifying when query using non-monotonic operators is monotonic. Indeed, one interesting approach is to allow the user or application to use non-monotonic constructs (as negation) but exclusively to write monotonic queries. Obviously, such approach avoids the loss of expressive power of queries and transformation in the case of online $\mathcal{M}$-Trace. Unfortunately, this approach is practically attractive only if the compiler/optimizer is capable of recognizing monotonic queries, and thus warning the user when a certain query is non-monotonic and thus can not be used in a continuous transformation. Unfortunately, deciding whether a query is monotonic can be computationally intractable, because depending not only on query but also on $\mathcal{M}$-Trace data, which can be obvious to the user but not the optimizer.

A better approach is to introduce new monotonic operators to extend the power of the query language. In this sense, we have use in our framework the recursion constructs $< Q >$ allowing to pattern to use and call a query. This recursive construct is monotonic if the called query uses only monotonic operator (all operator except {or, opt, without}) and its extends the power of our language to enable the expression of large class of queries. However, it is not clear whether our language with recursion are capable of expressing the class

of all monotonic queries. Although in the practice, most of queries which are monotonic can be easily expressed with recursion, we do not have yet a general answer for this interesting theoretical question. We will leave this question for later investigations, since it is not of urgent practical importance, given that, our experience in several projects shows that complex and non monotonic queries are often exploited over offline $\mathcal{M}$-Traces (by an analyst), and non-monotonic queries have not proven very useful for online transformations (to a programmer or the tracked user himself). In this paper, we follow a very practical approach allowing expressive queries for offline $\mathcal{M}$-Traces and less expressive queries for online ones (due to need to more fast and efficient online processing). This approach allows to our $\mathcal{M}$-Trace-based languages to deliver much higher levels of expressive power, not only in theory, but also in practice, as demonstrated in many applications implemented based on our framework.

So, there are many others questions left unanswered. The most interesting of them is when considering the semantics of a query and transformation languages over infinite $\mathcal{M}$-Traces. Indeed, querying $\mathcal{M}$-Traces may be developed with some assumptions:

1. We can assume that the current $\mathcal{M}$-Trace contains *complete information* about the observation and define semantics of queries with respect to the current (finite) $\mathcal{M}$-Trace. This approach corresponds to the closed-world assumption (WCA) semantics for queries.

2. Alternatively, we can treat the current $\mathcal{M}$-Trace as a finite prefix of infinite $\mathcal{M}$-Trace. In this setting the semantics of queries is defined with respect to infinite extensions (completions) of the current $\mathcal{M}$-Trace and is similar to the open-world assumption[13] (OWA).

In this paper we have restricted our attention to the closed-world assumption semantics only. This restriction postulates that, for the purpose of query answering, the only data values and time instants that exist are those present in the $\mathcal{M}$-Trace. All the quering and transforming techniques are developed with this restriction in mind. However, to bring more closer our framework to the real semantics associated with ontologies, we should discuss the implication of relaxing the WCA restriction.

Foremost, infinite $\mathcal{M}$-Traces notion is different from online and off-line $\mathcal{M}$-Trace ones. Infinite $\mathcal{M}$-Trace can be online or off-line, i.e. having only information to be completed (during evaluation) about observed elements and attributes (offline case) and such incomplete information may be evolves over time (online case). So far, we only focused on *finite* $\mathcal{M}$-Traces in both case. However, the alternative to this approach is that finite $\mathcal{M}$-Traces can be considered to be *finite parcel* among an infinity $\mathcal{M}$-Traces. Queries and transformations are then evaluated with respect to the infinite $\mathcal{M}$-Traces, by using the same semantics

[13]The open-world assumption, howerver, is assumed only for the following $\mathcal{M}$-Tracessets *extensions*: observed elements, relations, attributes); the observed element that describe observations in the current $\mathcal{M}$-Trace are still considered to contain complete information about those observations (i.e. define at least its timestamps values.

defined in section 6.2. The sole difference is that an infinite observed elements with an infinite attributes are allowed. However, as only finite portion of the $\mathcal{M}$-Trace is available at a particular (finite) point of time, we need to define answers to queries with respect to possible completions of the prefix to a infinite $\mathcal{M}$-Trace.

Let $\mathbb{T}^r$ be a finite $\mathcal{M}$-Trace, $\mathcal{Q}_{\mathbb{T}^r} = (n, \vartheta_Q, P_\mathcal{M})$ a query and ψ a substitution. We say that

- ψ is a *potential answer* for $\mathcal{Q}_{\mathbb{T}^r}$ with respect to $\mathbb{T}^r$ *if there is an infinite completion* $\mathbb{T}_i^r$ *of* $\mathbb{T}^r$ *such that* $\models_{\mathbb{T}_i^r} \psi(P_\mathcal{M})$.
- ψ is a *certain answer* for $\mathcal{Q}_{\mathbb{T}^r}$ with respect to $\mathbb{T}^r$ *if for all infinite completions* $\mathbb{T}_i^r$ *of* $\mathbb{T}^r$ *we have* $\models_{\mathbb{T}_i^r} \psi(P_\mathcal{M})$.

12 Conclusion

This paper provides a general and formal framework to exploit traces modeled by ontology-like model describing an observation of users interacting with computers. We have present a general architecture abstracting the notion of Trace-Based Systems as KBS reasoning about such traces. Firstly, we have began to formalise the notion of $\mathcal{M}$-Trace in the context of offline exploitation. Then, we have specified languages to query and transform such $\mathcal{M}$-Traces through patterns and template rules. The semantics described for $\mathcal{M}$-Traces and associated languages has been extended to online exploitation by defining the notion of online $\mathcal{M}$-Trace and continuous queries and transformations. Finally, we have describe an implementation of our languages by defining a translation into a deductive database and the rule language datalog for a robust and efficient framework realisation.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution, 2003.

[3] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[4] Ming-Syan Chen, Jong Soo Park, and Philip S. Yu. Efficient data mining for path traversal patterns. *IEEE Trans. on Knowl. and Data Eng.*, 10(2):209–221, 1998.

[5] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A language for analyzing transactional data streams. *ACM Trans. Program. Lang. Syst.*, 26(2):301–338, 2004.

[6] Richard Cyganiak. A relational algebra for SPARQL. 2005.

[7] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[8] Prud'hommeaux Eric and Seaborne Andy. Sparql query language for rdf, Junuary 2008.

[9] A. Foss, W. Wang, and O. R. Zaiane. A non-parametric approach to web log analysis. *Proceedings of Workshop on Web Mining in First International SIAM Conference on Data Mining, Chicago.*

[10] Johannes Gehrke and Praveen Seshadri. Towards sensor database systems. pages 3–14, 2001.

[11] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[12] Thanaa M. Ghanem. Incremental evaluation of sliding-window queries over data streams. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):57–72, 2007. Member-Hammad,, Moustafa A. and Member-Mokbel,, Mohamed F. and Senior Member-Aref,, Walid G. and Senior Member-Elmagarmid,, Ahmed K.

[13] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[14] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 658–669, New York, NY, USA, 2005. ACM.

[15] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 48–57, New York, NY, USA, 2003. ACM.

[16] Kristina Höök. Evaluating the utility and usability of an adaptive hypermedia system. In *IUI '97: Proceedings of the 2nd international conference on Intelligent user interfaces*, pages 179–186, New York, NY, USA, 1997. ACM.

[17] Alfred Kobsa. Generic user modeling systems. *User Modeling and User-Adapted Interaction*, 11(1-2):49–63, 2001.

[18] Jurgen Kramer. *Continuous Queries over Data Streams – Semantics and Implementation*. PhD thesis, In GI-Edition-Lecture Notes in Informatics (LNI): Ausgezeichnete Informatikdissertationen 2007, 2007.

[19] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 492–503. VLDB Endowment, 2004.

[20] Yue-Shi Lee and Show-Jane Yen. Incremental and interactive mining of web traversal patterns. *Inf. Sci.*, 178(2):287–306, 2008.

[21] Mark Levene and George Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, London, UK, 1999.

[22] Bing Liu. *Web Data Mining, Exploring Hyperlinks, Contents and Usage Data*. Springer, December 2006.

[23] F. Masseglia, P. Poncelet, and M. Teisseire. Using data mining techniques on web access logs to dynamically improve hypertext structure. *SIGWEB Newsl.*, 8(3):13–19, 1999.

[24] Bamshad Mobasher, Honghua Dai, Tao Luo, Yuqing Sun, and Jiang Zhu. Integrating web usage and content mining for more effective personalization. In *EC-Web*, pages 165–176, 2000.

[25] David Peterson, Ashok Malhotra, C. M. Sperberg-McQueen, and Thompson Henry S. W3c xml schema definition language (xsd) 1.1 part 2: Datatypes, Junuary 2009.

[26] Wolfgang Pohl, Alfred Kobsa, and Oliver Kutter. User model acquisition heuristics based on dialogue acts. In *In International Workshop on the Design of Cooperative Systems*, pages 471–486, 1995.

[27] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Seq: A model for sequence databases. pages 232–239, 1995.

[28] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274, New York, NY, USA, 2004. ACM.

[29] ISO Standards. Representation of dates and times. iso 8601:2004, 2004.

[30] Adelheit Stein, Jon Atle Gulla, and Ulrich Thiel. Making sense of users' mouse clicks: Abductive reasoning. In *and Conversational Dialogue Modeling.. Sixth International Conference on User Modeling. Chia*, pages 89–100. Springer Wien, 1997.

[31] Mark Sullivan and Andrew Heybey. Tribeca: a system for managing large databases of network traffic. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[32] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass, editors. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[33] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330, New York, NY, USA, 1992. ACM.

[34] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[35] Timo Weithoner, Thorsten Liebig, and Gunther Specht. Storing and querying ontologies in logical databases, September 2003.

[36] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks, 2002.

13 APPENDIX I: Datalog

To make the report self-contained, we will briefly review notions of Datalog. The reader can consult [7] for further details and proofs.

13.1 Syntax of Datalog

- A *term* is either a variable or a constant.
- A *predicate formula* is an expression $p(x_i, ..., x_n)$ where p is a predicate name and each x_i is a variable[14].
- An *equality formula* is an expression $t_1 = t_2$ where t_1 and t_2 are terms.
- An *atom* is either a predicate or an equality formula.
- A literal is either an atom (a positive literal L) or the negation of an atom (a negative literal $\neg L$).
- A *rule* is an expression of the form

  $$L :- L_1, ..., L_n$$

  where L is a predicate formula called the *head* of the rule and the sequence of literals $L_1, ..., L_n$ is called the body of the clause. Note that may assume that all heads of rules have only variables by adding the respective equality formula to its body.
- A *Datalog program* is a finite set of Datalog rules.
- Let Π be a Datalog program. A rule having no variables is called a *ground rule*. A ground rule with empty body is called a *fact*.
- The set of facts occurring in Π, denoted $facts(\Pi)$.
- A predicate is *extensional* if it occurs only in facts; otherwise it is called *intensional*.
- A variable X occurs positively in a rule r if and only if X occurs in a positive literal L in the body of r such that:
  1. L is a predicate formula;
  2. if L is $X = c$ then c is a constant;
  3. if L is $X = Y$ or $Y = X$ then Y is a variable occurring positively in r.

[14] This is our assumption in this report that a predicate formula only contains variables, but in general case it is also possible to have constants.

A Datalog rule r is said to be safe if all the variables occurring in the literals of r (including the head of r) occur positively in r. A Datalog program Π is safe if all the rules of Π are safe. The dependency graph of a Datalog program Π is a digraph (N, E) where the set of nodes N is the set of predicates that occur in the literals of Π, and there is an arc (p_1, p_2) in E if there is a rule in Π whose body contains predicate p_1 and whose head contains predicate p_2.

A Datalog program is said to be recursive if its dependency graph is cyclic, otherwise it is said to be non-recursive. A Datalog query is a pair (Π, L) where Π is a Datalog program and L is a predicate formula $p(x_1, ..., x_n)$ called the goal literal.

13.2 Semantics of Datalog

A *substitution* θ is a set of assignments $\{x_1/t_1, ..., x_n/t_n\}$ where each X_i is a variable and each t_i is a term. Considering a rule r, we denote by $\theta(r)$ the rule resulting from applying the substitution θ to the literals in r, i.e., the result of substituting the variable X_i for the term t_i in each literal of r. A substitution is ground if every term t_i is a constant.

A rule r in a Datalog program Π is true with respect to a ground substitution θ, if for each literal L in the body of r one of the following conditions is satisfied:

1. $\theta(L)$ $\in$facts(Π);
2. $\theta(L)$ is an equality, $c = c$ where c is a constant;
3. $\theta(L)$ is a literal of the form $\neg p(c_1, ..., c_n)$ and $p(c_1, ..., c_n)$ $\in$ facts(Π);
4. $\theta(L)$ is a literal of the form $\neg(c1 = c2)$ and c_1 and c_2 are distinct constants.

The *meaning* of a Datalog program Π, denoted facts$^*(\Pi)$, is the database resulting from adding to the initial database of Π as many new facts of the form $\theta(L)$ as possible, where θ is a substitution that makes a rule r in Π true and L is the head of r. Then the rules are applied repeatedly and new facts are added to the database until this iteration stabilizes, i.e., until a fixpoint is reached [7].

Given a Datalog query $Q = (\Pi, L)$ and the initial database D = facts(Π). The *answer* to Q over database D, denoted $ans_d(Q, D)$, is a set of substitutions defined as $ans_d(Q, D) = \{\theta \mid \theta(L) \in \text{facts}^*(\Pi)\}$.

14 APPENDIX II: Properties and Complexity of Evaluating $\mathcal{M}$-Trace Patterns

A fundamental issue in every query language is the complexity of query evaluation and, in particular, what is the influence of each component of the language in this complexity. In this section, we address these issues for $\mathcal{M}$-Trace pattern expressions.

As it is customary when studying the complexity of the evaluation problem for a query language, we consider its associated decision problem. We denote this problem by *Evaluation* and we define it as follows:

- INPUT : An $\mathcal{M}$-Trace $\mathbb{T}^r$, a pattern P and a substitution ψ.
- QUESTION : Is $\psi \in [\![P]\!]_{\mathbb{T}^r}$?

We start this study by considering the fragment consisting of $\mathcal{M}$-Trace pattern expressions constructed by using only AND operators. This simple fragment is interesting as it does not use the two most complicated operators in $\mathcal{M}$-Trace Query, namely OR and OPT. Given a $\mathcal{M}$-Trace $\mathbb{T}^r$, a pattern P in this fragment and a substitution ψ, it is possible to effciently check whether $\psi \in [\![P]\!]_{\mathbb{T}^r}$ by using the following algorithm.

- First, for each pattern p in P, verify if $\models_{\mathbb{T}^r} p$ i.e. $\mathbb{T}^r$ entails p.
- If this is not the case, then return false.

Thus, we conclude that:

Theorem 1. *Evaluation $[\![P]\!]_{\mathbb{T}^r}$ can be solved in time $\mathcal{O}(|P| \cdot |O \cup V|)$ for $\mathcal{M}$-Trace pattern expressions constructed by using only AND operator.*

Theorem 2. *Evaluation is NP-complete for a general pattern expressions constructed by using only the basic pattern with AND and OR operator.*

We will continue this study by adding to the above fragment the OR operator and then more complicated OPT operator.

Theorem 3. *Evaluation is PSPACE-complete for a general pattern expressions constructed by using only the basic pattern with OR and OPT operator.*